Audio Frequency Analyzer

A Project by

Gerrald Carlo A. Mateo
Zarah Katrina E. Pulmano
Maria Mae Fleur M. Villanueva

Submitted to

Luisito L. Agustin
Instructor, ELC 152

In Partial Fulfillment of the Requirements for the Course
ELC 152: Signal Processing

Department of Electronics, Computer and Communications Engineering
School of Science and Engineering
Loyola Schools
Ateneo de Manila University
Quezon City, Philippines

October 2010

Abstract

An Audio Frequency Analyzer that displays an audio sample in the frequency spectrum is implemented using C++. First, it is programmed to record 16-bit audio samples with a sampling rate of 8 kHz. Another class is created to provide functions that get the Discrete Fourier Transform (DFT) of the audio samples. Finally, a graphical user interface is developed to display the frequency spectrum obtained from the DFT computations. Note that the recording and graphing of the frequency spectrum is done continuously and by blocks, where each audio block consists of 1024 samples.

Acknowledgments

Our sincerest acknowledgement to the following people who contributed to the success of this project:

First, we would like to thank our professor, Mr. Luisito Agustin for allowing us to develop an understanding of our project, for always making himself available for consultations and for directing us towards the accomplishment of this project.

Special thanks goes to our classmates Mr. Ricson Chua and Mr. Francis Tiausas, for giving us advice and for helping us solve the problems that we encountered through the process.

Of course, we extend our gratitude to our family, especially our parents, for their unconditional support and encouragement in our pursuits. Without them, we could not have gone to this point in our life in the first place.

All our regards and blessings, we offer to all others who supported us in any respect during the completion of the project.

Table of Contents

1. Introduction 6
1.1. Project Objectives.....6
1.2. Project Scope and Limitations.....6
1.3. Overview of Components.....6
2. XMCS Player/Recorder ….....8
2.1. XMCS Class.....8
2.1.1. XMCS() Constructor.....….8
2.1.2 XMCS Record function.....9
2.1.3 XMCS Play function ….....9
2.1.4 XMCS Save and WriteBlockFromBuffer function.....10
2.1.5 XMCS AccessBlocks function.....10
2.1.6 Members.....10
2.2. Utility Class 12
3. Fourier Class.....13
3.1. Theoretical Background.....13
3.1.1. Decimation-in-Time Algorithm.....13
3.1.2. Bit-Reversal.....15
3.1.3. Fourier Class.....15
3.1.3.1. Members.....15
3.1.3.2. Functions.....16
4. User Interface: Implementation ….....17
4.1. audiofrequencyanalyzerfrm.h..... 18
4.2. audiofrequencyanalyzerfrm.cpp..... 19
4.2.1. Program Flow.....19
5.User's Manual..... 21
5.1. CDROM Contents.....21
5.2. Software Features.....22

5.3. Using the Software....22
6. Tests ….....25
6.1. LabVIEW....25
6.2. Decimation-in-Time FFT function vs. DFT Test Code....25
6.3. Constant Input.... 28
6.4. Sinusoidal Waves with Different Frequencies....28
6.5. Monophonic, Polyphonic, and True Tones....33
6.6. Theoretical vs Actual Audio Blocks Processed....35
6.7. Record Testing....35
7. Results and Recommendations....37
Appendix....38
Source Code....38
A.1. Source Files....38
A.1.1. audiofrequencyanalyzerfrm.cpp….....38
A.1.2. audiofrequencyanalyzerfrm.h....41
A.1.3. audiofrequencyanalyzerapp.cpp....42
A.1.4. audiofrequencyanalyzerapp.h....43
A.1.5. xmcs.cpp....44
A.1.6. xmcs.h....59
A.1.7. utility.cpp....62
A.1.8. utility.h....65
A.1.9. fourier.h....66
Bibliography 69

1. Introduction

1.1. Project Objectives

The main objective in this project is to create an Audio Frequency Analyzer which is an application that records and displays successively the frequency spectrum of audio samples. The recording of the audio samples and the displaying of the frequency spectrum is intended to be done by blocks where the length of the audio blocks is fixed to 1024 samples while the number of audio blocks to be processed is to be specified by the user.

1.2. Project Scope and Limitations

The recording part of the project supports only 16-bit XMCS audio and the sampling rate is fixed to 8 kHz. The recording and displaying of its Discrete Fourier Transform is automatically looped where the number of loops is equal to the number of audio blocks specified by the user. The recording could not be paused and then resumed. It will also not stop until the recording has not reached the specified number of loops set by the user.

1.3. Overview of Components

In the development of the Audio Frequency Analyzer, the following classes are utilized:

- the XMCS class that provided the recording functionality that is needed to collect audio samples that are to be accessed for the DFT computation;

- the Utility class that contained the functions needed to convert the 16-bit audio samples to a type that the DFT computation can handle;

- the Fourier class that is used in executing the DFT computations;

- and finally, the audiofrequencyanalyzerfrm class for the development of the graphical user interface of the program.

2. XMCS Player / Recorder

The parts used from the XMCS Player / Recorder project from 2007 is discussed in this chapter. The two classes used from this project are the XMCS class and the Utility Class. In addition, the main functions used are the record function, play function, and save function. A new function is also added to this class. It is used to access the audio blocks to be able to transfer them for FFT computation later on. The changes made to the original files are also indicated in this chapter. Some minor changes are also done to the record and save functions for maximized compatibility with the graphical user interface and checking. Parts that were not directly used or altered will not be discussed anymore because it is already expounded in the XMCS Player / Recorder documentation (Baterina and Macalintal, 2008).

2.1 XMCS Class

This class is used because it has audio recording and playing capabilities which are needed for the audio frequency analyzer. Furthermore, it is able to provide direct access to the audio blocks being recorded into memory which are needed for the FFT computation. The audio handling parts of the analyzer are taken care of by this class.

2.1.1 XMCS() Constructor

This constructor is the one that sets the default values to be used if the user does not opt to change the settings of the audio format. For the frequency analyzer, the user is not asked to set these settings so the default values indicated here are the ones used throughout the project. Important parameters are bits per sample and samples per second. These are set to 16 and 8000 respectively.

2.1.2 XMCS Record function

This function is a vital part of this project because it is the function that starts the whole process. The samples used for the FFT computation is taken from output of the recorder. Therefore, the output of the recorder is the input of the FFT. For this project, the audio block size for recording is set to 1024 (a power of 2) samples so that it can be fully accommodated and processed by the FFT function. Details on the FFT will be discussed in chapter 3.

Audio is recorded directly from an audio input device and is stored into temporary buffers and then into a linked list *(audioBuffer)*. To be able to do this, the audio input device is first opened for recording. After which, the buffers that will be filled with the audio samples from the input device are prepared.. Two temporary buffers are used so that there is always an available buffer to store the data once one of the buffers is filled. If a buffer is filled, the contents are automatically transferred to the end of *audioBuffer*. After it is successfully copied, the buffer is immediately cleared for the next set of audio samples.

Originally, the record function was intended for console applications. This made it possible to interrupt recording at any time. However, since the audio frequency analyzer is a windows application, the recording cannot be interrupted without the use of threads. To address this problem, the recording function was altered to stop after reaching 1024 samples. In the process, the buffer size was changed to 1024 samples with 16 bits per sample or simply 2048 bytes. Once the recording is halted, the buffer contents are transferred to *audioBuffer* and the audio input device is closed. A new recording will automatically overwrite the buffer contents from the previous recording.

2.1.3 XMCS Play function

The play function is not featured in the final executable application of the analyzer. However, it is important for the development of the project because it is used to check if the recorder successfully recorded the audio samples. It is also useful in checking if the buffer size was properly allocated by the recorder.

The play function is capable of playing from an XMCS file or from previously recorded data. However, only playing from memory is utilized in the analyzer development. Essentially, in the play function, the audio blocks taken from the linked list is played through an audio output device. Several buffers are also used to avoid losing the data that are being extracted from *audioBuffer.*

2.1.4 XMCS Save and WriteBlockFromBuffer function

The save function is the primary function used to check if the data accessed from *audioBuffer* correct. It is used to directly access the contents of *audioBuffer.* It makes use of a private function, the *WriteBlockFromBuffer* function to organize the format of the file. It was tweaked such that instead of saving the samples as binary or hexadecimal, the samples are saved into a file as signed decimal numbers. These two functions are crucial in the development of the new function in this class. They were combined to create the AccessBlocks function.

2.1.5 XMCS AccessBlocks function

This is a new function added to XMCS to be able to transfer the audio blocks into a global linked list of integers. By doing so, the audio samples are easily accessed by the FFT function for computations. The function is used to traverse the *audioBuffer* contents after recording. While traversing *audioBuffer,* the audio blocks stored in it are being stored into another linked list (*dataList*). The new linked list is a list of integers so it is easier to access compared to a list of pointers. The structure of this function is basically the combination of the private function *WriteBlockFromBuffer* and the save function.

2.1.6 Members

In this section, the key members used for the analyzer are described. Old and new members of the XMCS class are included in this section.

list<char> audioBuffer*

This is a private member which is used to store the contents of the temporary buffers.

list<int> dataList

This is a public member which stores the audio blocks and makes it accessible to the FFT function and the GUI (cplusplus.com, 2010).

list<int>::iterator it

This is a public member used to traverse the global list to be able to convert the list of integers into arrays for FFT. It is used in the GUI for traversing the currently recorded audio samples.

bool Save(string filename)

This is a public function that saves the audio samples into a file. The data representation of the samples can either be binary of hexadecimal but for the analyzer, it was used to save decimals.

bool Play()

This is a public function that plays the audio recorded from memory or audio from an XMCS file.

bool Record()

This is a public function that records audio and stores it into buffers and later into a linked list. Any new recording overwrites the audio currently stored.

void AccessBlocks()

This is a public member used to access the audio blocks as integers for the FFT.

2.2 Utility Class

This class was not used for the later part of the analyzer development. The primary function of this class is to convert integers into binary or hexadecimal data types. It was used at first when the audio blocks were being accessed through save and *WriteBlockFromBuffer* functions. However, when the recording part of the analyzer was integrated with the FFT, the audio samples were left as decimals so this class is unused in the final implementation.

3. Fourier Class

3.1. Theoretical Background

3.1.1. Decimation-in-Time Algorithm

Given below is the equation for the Discrete Fourier Transform,

$$F[n]=\sum_{k=0}^{N-1} f[k]e^{-j\frac{2\pi}{N}nK} \tag{3.1}$$

where $W_N^{nK}=e^{-j\frac{2\pi}{N}nK}$ is referred to as the twiddle factor.

We can further break up this equation by grouping together its even and odd components as shown in the following equation.

$$F[k]=\sum_{k=0}^{N-1} f[n]W_N{}^{nk} \tag{3.2.1}$$

$$F[k]=\sum_{m=0}^{\frac{N}{2}-1} f[2\mathrm{m}]W_{\frac{N}{2}}{}^{mk}+W_N{}^{k}\sum_{m=0}^{\frac{N}{2}-1} f[2\mathrm{m}+1]W_{\frac{N}{2}}{}^{mk} \tag{3.2.1}$$

$$F[k]=G[k]+W_N^k H[k] \tag{3.23}$$

By continuously dividing up or decimating the sample sequence into odd and even sub-sequences, we can turn the complex DFT equation into a set of 1-point DFT's such that our N-point DFT becomes a combination of these smaller and simpler DFT's. This is the goal of the decimation-in-time algorithm. Note however that the decimation only works if the number of our sampling points is a power of 2. Meanwhile, the following properties are useful to further simplify the DFT equation.

Periodicity Property: $W_N{}^{k+N}=W_N^k$ (3.3.1)

Symmetry Property: $W_N{}^{k+N/2}=-W_N^k$ (3.3.2)

The symmetry property is used to obtain a function as shown in eq. 4.

$$F[k+N/2]=G[k]-W_N^k H[k] \tag{3.4}$$

Thus, the DFT of next half of the N samples can be obtained by using the same set of computations from the the first half, and then just changing the operation to subtraction. For illustrative purposes, the figure below shows the computation of an 8-point DFT. (ee.caltech.edu, 2010).

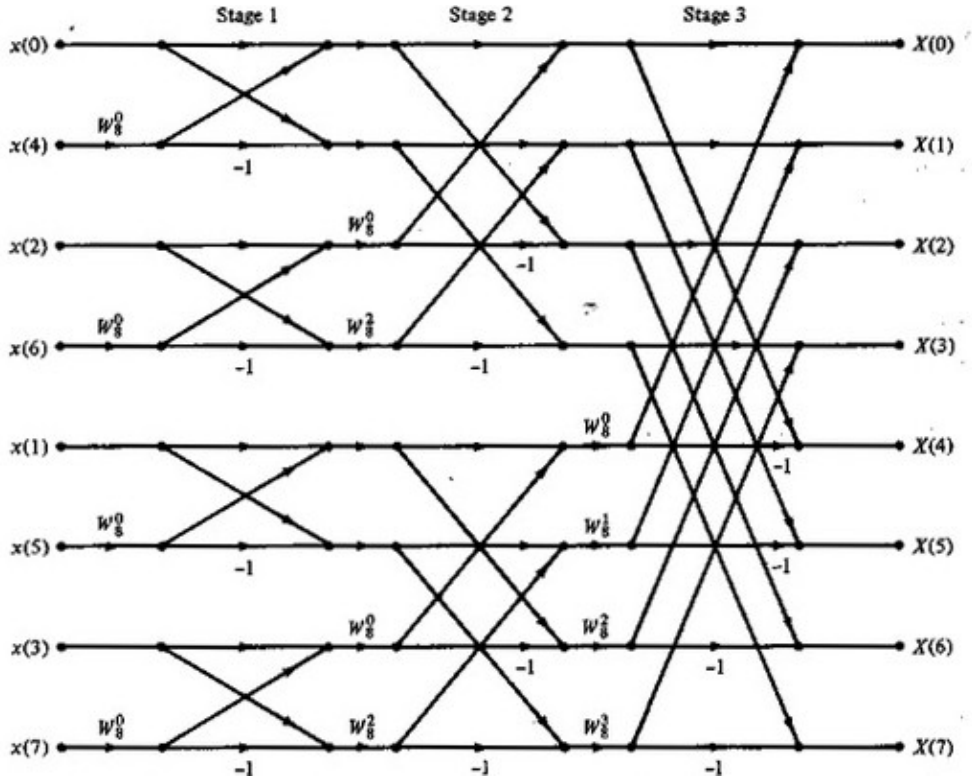


Figure 3.1 Eight-point decimation-in-time FFT algorithm.

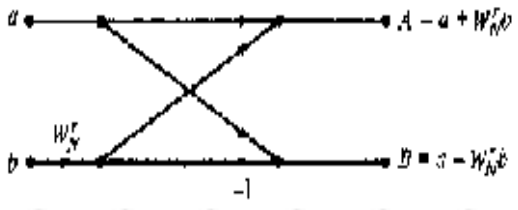
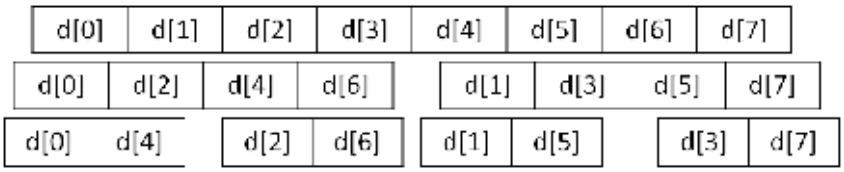


Figure 3.2. Basic butterfly computation in the decimation-in-time FFT algorithm.

3.1.2. Bit-Reversal

Since the decimation-in-time algorithm involves rearranging our sample sequence, it needs another algorithm that should put the samples back to their original sequence. Taking for example an 8-point block of data with indices 0-7, using FFT using Decimation-in-Time Algorithm, we can break down the whole block into the following:



Note however that the indices of the new sequence that was produced, when converted to their binary forms, are just the bit reversed form of the original sequence. Thus, all calculations can be done in position if the original data sequence is shuffled already into the correct order (the bit reversed form) before passing it into the FFT algorithm.

3.1.3. Fourier Class

For this audio frequency analyzer project, a Fourier class, whose members and functions are enumerated below, is created to provide functions that shall execute the FFT computation of our audio samples.

3.1.3.1. Members

The Fourier class includes the following members:

complex<double> data_array[1024];

This is a member of the class that is used to store data samples so that it is readily processed by the functions inside the Fourier class.

int block_length;

This is a member of the class that is used to store the block length of the data sequence.

3.1.3.2. Functions

The Fourier class includes the following functions:

void set (int *data[],* int *length*);

This function sets or readies the data block it receives as a complex type array. This function requires two parameters: the *data* array which is the block containing the input for the FFT computation and the *length* of the data block or the number of sampling points that will be computed. Before running or calling the other functions in this class, it is necessary that this function is called first.

void FFT_function(void);

This function is called to get the Fast Fourier transform of the complex data array that is set using the set() function.

void bit_reverse_function(void);

This function performs the bit reverse function that is needed before performing the main FFT computation. Note, however, that there is no need to call this function anymore because it is automatically performed when the FFT_function() is called.

void magnitude(double mag[]);

This function computes and stores into an array the magnitudes of each point in the FFT computation. It requires a parameter which is a double type array that will be used as the storage for the magnitude computations.

4. User Interface: Implementation

The user interface of the Audio Frequency Analyzer program is designed using the wxWidgets cross-platform C++ GUI toolkit (Koga, 2009). The graphical user interface of the program allows to analyze an audio recorded from the sound card and count the number of audio blocks being processed. The window of the GUI is composed of the output field, the control button and the graph field as shown Fig. 4.

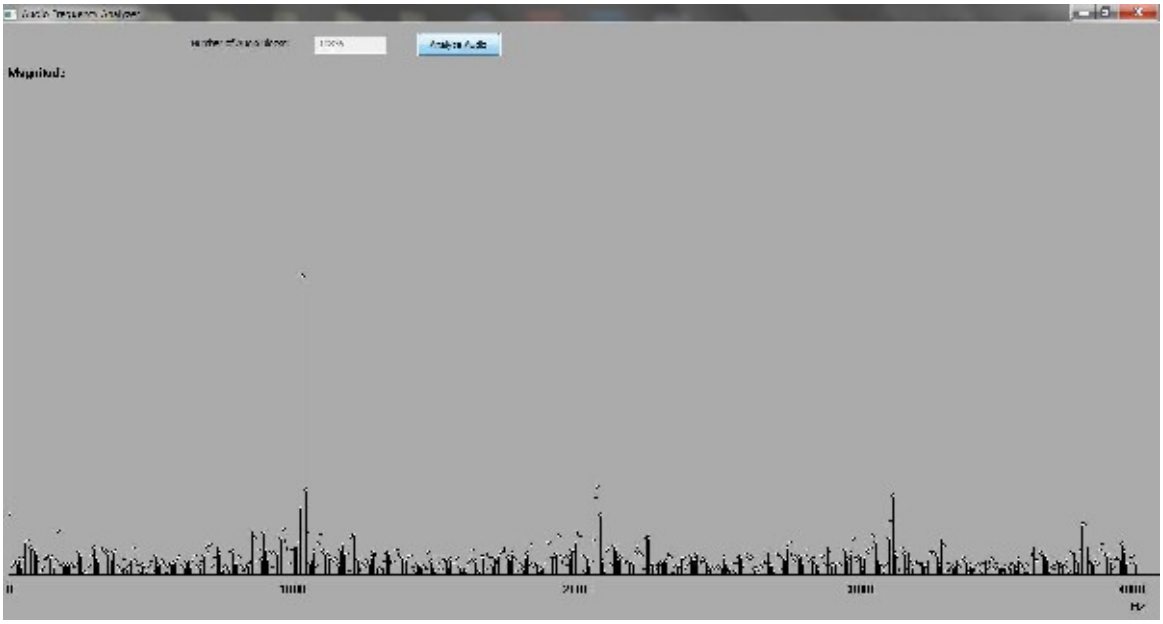


Fig. 4.Audio Frequency Analyzer GUI

The source code involved in the GUI are the following: audiofrequencyanalyzerapp.h, audiofrequencyanalyzerapp.rc, and audiofrequencyanalyzerapp.cpp, which are used to implement the main program instructed from the audiofrequencyanalyzerfrm.h and audiofrequencyanalyzerfrm.cpp that are necessary to call the functions and execute them through the buttons placed on the audiofrequencyanalyzerfrm.wxform. The program is supported by the XMCS class and Fourier class. The source code for these classes are completed with xmcs.cpp, xmcs.h, utility.cpp, utility.h and fourier.h.

4.1. audiofrequencyanalyzerfrm.h

audiofrequencyanalyzerfrm.h is the location for declaring global variables and constructors for the program. The wxFrame platform is used in order to utilize the size of of the window.

Several wx classes are used in this program such as

```
#include <wx/dcclient.h>
#include <complex>
```

All the drawing techniques for the GUI are contained inside the dcclient.h (Smart et al, 2010). The complex class is included for the Fourier class.

The variable declared as public inside the audiofrequencyanalyzerfrm class are as follows:

```
XMCS audio;
void WxButton1Click(wxCommandEvent& event);
```

The void function above is automatically inserted when the button is placed on the ~Frm.wxform. XMCS audio is a constructor for the XMCS class. In this manner, the variable audio is able to call the functions inside the XMCS class.

The variable declared as private inside the audiofrequencyanalyzerfrm class are predefined strings for the wxWidgets platform such as

```
wxClientDC *graph;
wxTextCtrl *countAudioBlocks;
wxButton *analyzeAudio;
wxStaticText *numberOfAudioBlocks;
```

Similarly, the constructors call on the functions inside the adjacent class.

4.2. audiofrequencyanalyzerfrm.cpp

The class for the GUI is audiofrequencyanalyzerfrm. audiofrequencyanalyzerfrm.cpp is the location for most of the syntax of the program which calls all the functions from the necessary classes appended to it. This code is mainly composed of two important function. The output field is given by the `wxEdit` text field from the wxTextCtrl class and the application is enabled using wxButton1Click.

4.2.1. Program Flow

The event is started when the user clicks the "Analyze Audio" button shown in Fig. 4.2.2.

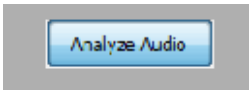


Fig. 4.2.2 Analyze Audio Button

The number of blocks being processed is shown in the output field read in "Number of Audio Blocks:" as shown in Fig. 4.2.1.

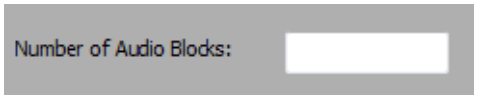


Fig. 4.2.1 Input Field

The default size of the audio block is set to 1024 samples.

The while inside the program is composed of several for loops. This while loop is used to allow the program to process continuously.

The first function to be called inside the while loop is the record function by

```
audiofrequencyanalyzerfrm::audio.Record();
```

The variables for the FFT function is declared next to the record function because some variables are needed for the first for loop which has the linked list and immediately the FFT is computed by taking the data from the second for loop, that is the list provided by the XMCS class.

```
compute.set(data, np);
compute.FFT_function();
compute.magnitude(out);
```

The code above is setting the data to be computed and only the magnitude from the computed value are taken. Immediately after the computation, the graph is being shown. First, the x and y axes are drawn for reference purposes and then the values of the magnitudes of the FFT are shown which is inside the third for loop.

```
x = 5 + (i*1.3);
y = 650-(out[i]*2);
graph->DrawLine(x, 650, x, y);
graph->DrawCircle(x, y, 2);
```

In order to successfully loop the whole process, the source code is consisted of lines that clears the record, compute and draw functions. Inside the XMCS class, the `"Unload()"` function is able to remove the previous recorded sound once a new sound is recorded. Inside this source code, the `audio.dataList.clear();` clears the contents of the list before another data comes in, the graph`->Clear();` is prompted to clear the background immediately before it draws the x and y axes and the magnitudes of the FFT.

The labels for 0, 1000, 2000, 3000, and 4000 Hertz and the Magnitude are printed on the GUI using `graph->DrawText("the text", x-coordinate, y-coordinate).`

Counting the number of blocks processed is possible with the counter inside the source code with this syntax:

```
int ctr = 0;
ctr++;
countAudioBlocks->Clear();
*countAudioBlocks<<ctr;
```

The first line is placed before the while loop. That denotes where the counting should start and it ends with the second line placed right after the loop which graphs the FFT. This is where the counts starts at 1 and so on. The fourth line shows the number on the output field. The third line clears the previous number so that the output field shows increments of integer.

5. User's Manual

5.1. CDROM Contents

- <CD root directory>\
  - index.html\ - *homepage of project website*
  - software\
    - Output\
      - MingW\
        - audiofrequencyanalyzer.exe – *Audio Frequency Analyzer*
    - source_code\
      - xmcs.h – *XMCS class definition*
      - xmcs.cpp – *XMCS class implementation*
      - utility.h – *Utility class definition*
      - utility.cpp – *Utility class implementation*
      - fourier.h – *Fourier class definition*
      - audiofrequencyanalyzerfrm.h – audiofrequencyanalyzerfrm *class*
      - *definition*
      - audiofrequencyanalyzerfrm.cpp – audiofrequencyanalyzerfrm *class implementation*
      - audiofrequencyanalyzer.dev – *Project compilation w/ all classes included.*
      - audiofrequencyanalyzerapp.h – a*udiofrequencyanalyzerapp definition*
      - audiofrequencyanalyzerapp.cpp – *audiofrequencyanalyzer main program*
  - doc\
    - documentation.odt – *project documentation (.odt file)*
    - documentation.pdf – *project documentation (.pdf file)*

- progress_reports\ – *contains progress reports of group members*
  - mateo\
  - pulmano\
  - villanueva\
- proposal\ - *contains approved project proposal*
  - proposal.odt

5.2. Software Features

The XMCS Player / Recorder allows the user to do the following:

- Record audio
- Automatically display the frequency spectrum upon recording
- Modify the number of audio blocks to be recorded and display.

5.3. Using the Software

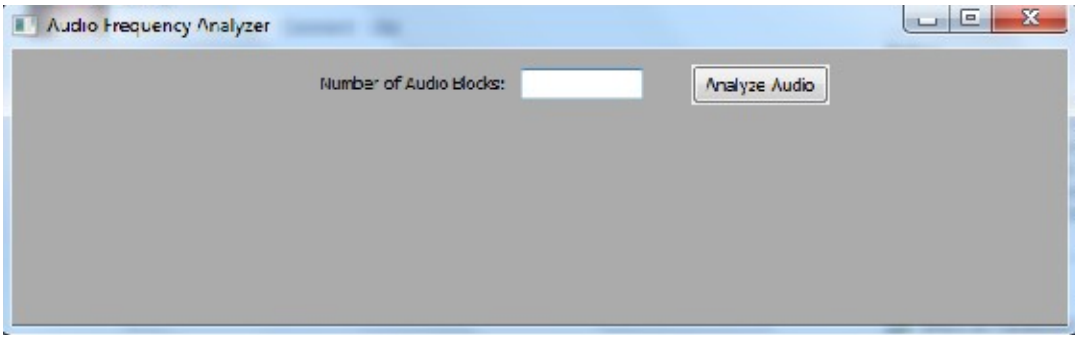


Fig. 5.3.1 Opening Screen

When the program is ran, an opening screen (Figure 5.3.1) is displayed. The user can choose to maximize the window to view the graph in a better resolution.

The user may then press the button *Analyze Audio* to start the recording and see a dynamic graph of the frequency spectrum of the audio recorded from the microphone. It is also important to note that the settings of the microphone being used is crucial in obtaining meaningful and accurate analysis. Having a very high microphone input volume would make the graph very noisy even if the surrounding environment is quiet as shown in Figure 5.3.2.

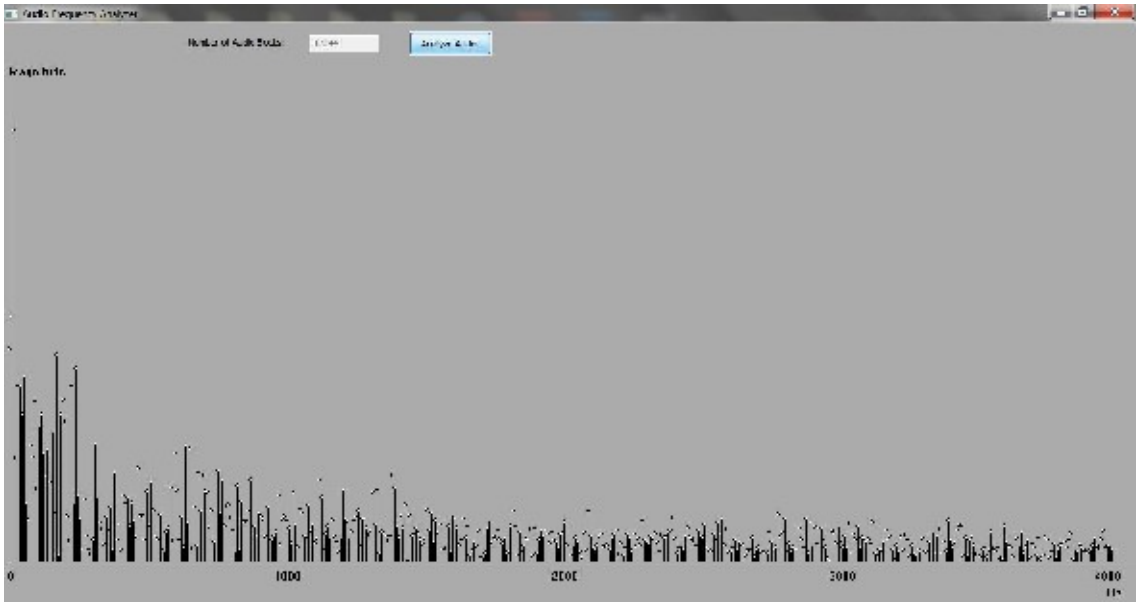


Fig. 5.3.2. Analyzed output with microphone volume set to 100% in a quiet room.

Moreover, boosting the microphone input will also make the resulting graph inaccurate. It is advisable to set the microphone boost setting to 0dB because setting it to 30dB (maximum boost with Windows 7 operating system) will make the graph consistently noisy as well. This result is shown in Figure 5.3.3.

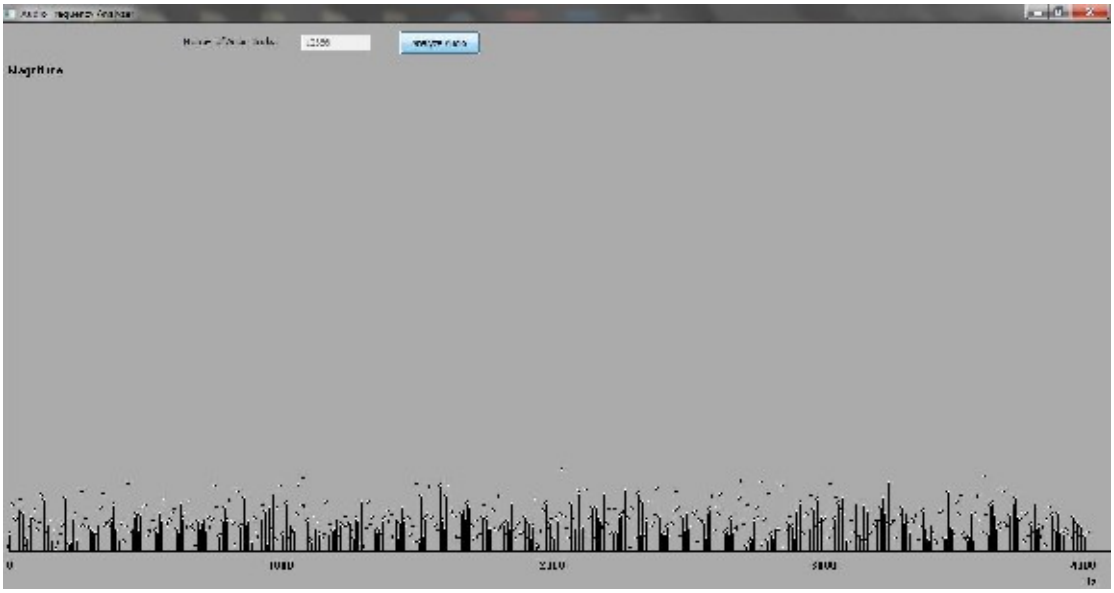


Fig. 5.3.3. Analyzed output with microphone volume set to 100% and microphone boost set to 30dB in a quiet room.

By testing the software several times, we concluded that the optimal microphone volume is 30% because it lowers the noise considerably and it makes the program more reactive to the audio played within the range of the microphone. The resulting graph in a silent room with volume at 30% is shown in Figure 5.3.4

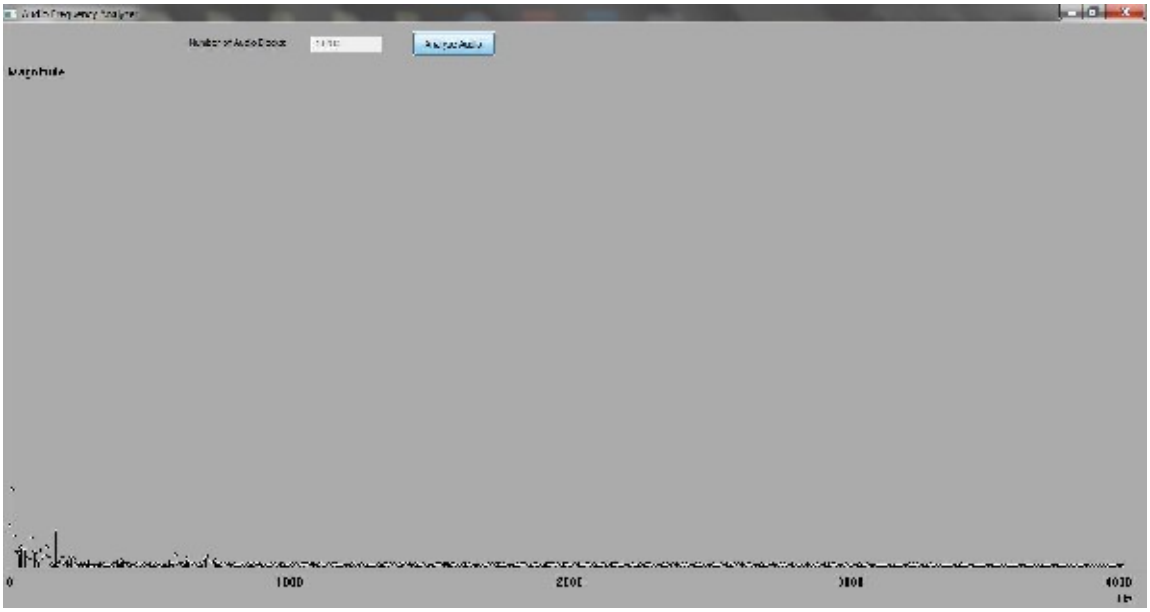


Fig. 5.3.4. Analyzed output with microphone volume set to 30%.

6. Tests

Following are tests done to verify the accuracy of the Audio Frequency Analyzer Project.

6.1. LabVIEW

For comparison purposes, a simple FFT system was developed using the LabVIEW environment, where the input is an 8-point sample with values from 0 to 7. The DFT computation results that were generated from the LabVIEW are then compared to those generated from the FFT decimation-in-time function code that was created. Note that the input sample sequence for the FFT function is just the same as our input in the Labview. As observed, both produced similar results.

6.2. Decimation-in-Time FFT function vs. DFT Test Code

Another test code which is patterned to the original DFT equation is also created to further check the accuracy of our decimation-in-time FFT function. We verified if the results generated using this code matches those that were obtained using the FFT function. The code is shown below.

```
for (int k=0; k< np; k++)           //loop to get the DFT for
{                                         all points from 0 to np
      for (int n=0; n< np; n++)     //loop where the DFT
      {                                   computation is done
            complex<double> temp;
            double m = 6.28318530717959*k*n/np;
            double q = cos(m);
            double r = sin(m);
            complex<double> W(q,-r);
            temp= data[n]*W;
            F[k]=F[k]+temp;         //increments for every loop to
                                        implement the summation
                                        process.
            W=(0,0);
      }
}
```

While the GUI is still being developed, results generated from the test code and the FFT function are first displayed in the command prompt. The number of inputs that it can process is however limited to 16 points only, where above this number, the window does not respond anymore. The codes are tried out using 8-point and 16-point input samples. The figure below shows the comparison for a 16-point sample sequence with inputs 0 to15.

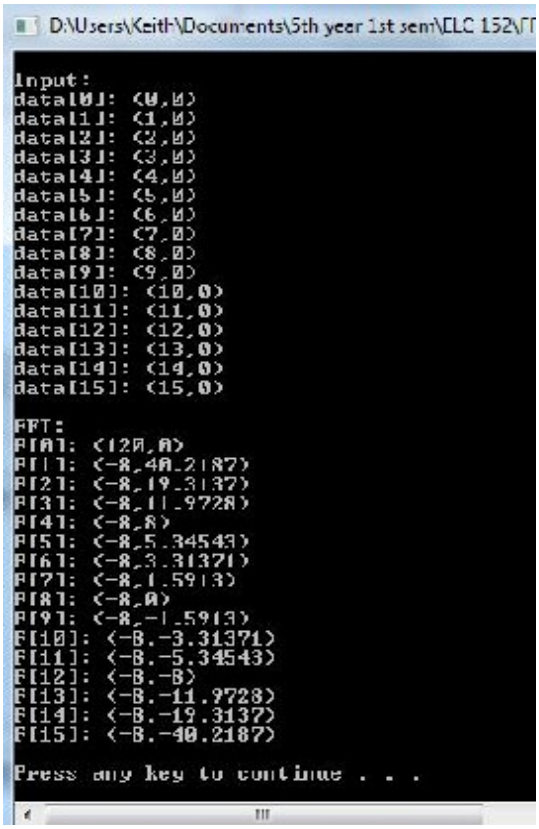


Figure 6.2.1. FFT code-generated results for a 16-point sample.

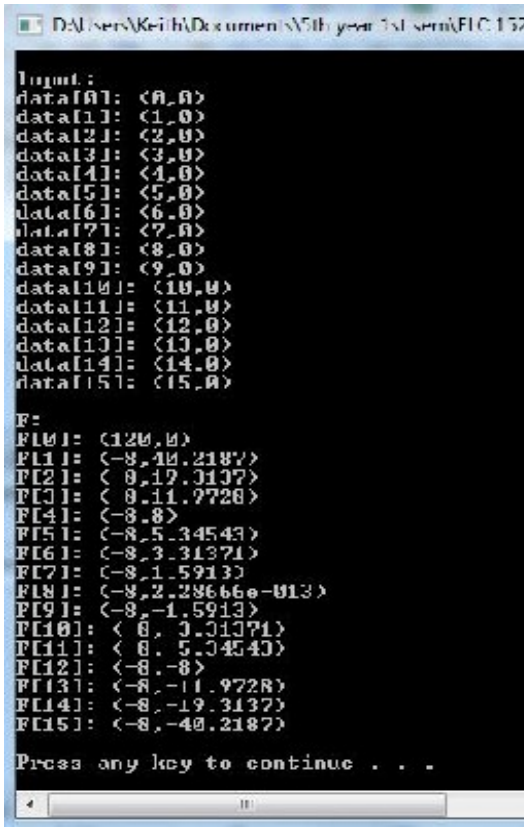


Figure 6.2.2. Test code-generated results for a 16-point sample

It can be seen from the two figures that the results generated are the same except for one point only, which is the middle point F[8]. For the FFT function, it produced a result of (-8, 0) while the test code had (-8, -2.28666e-13). However, since the difference is not that significant, we can say that our FFT function is fairly accurate. We also had a similar observation when the codes are tried out with 8-point samples. With 0 to 7 as the input values, the computation for the middle point F[4] in the original FFT function is (-4, 0) while for the test code, we had (-4, 3.38746e-14).

When the GUI was already created, we tried comparing both codes again by displaying their graphs for 1024 input samples with values from 0 to 1023. It can be observed from the figures below that the graphs are more or less the same. The two figures, however, are compressed versions of the original pictures captured, the reason why the points are not clear.

Figure 6.2.3. 1024-point frequency spectrum using the FFT function

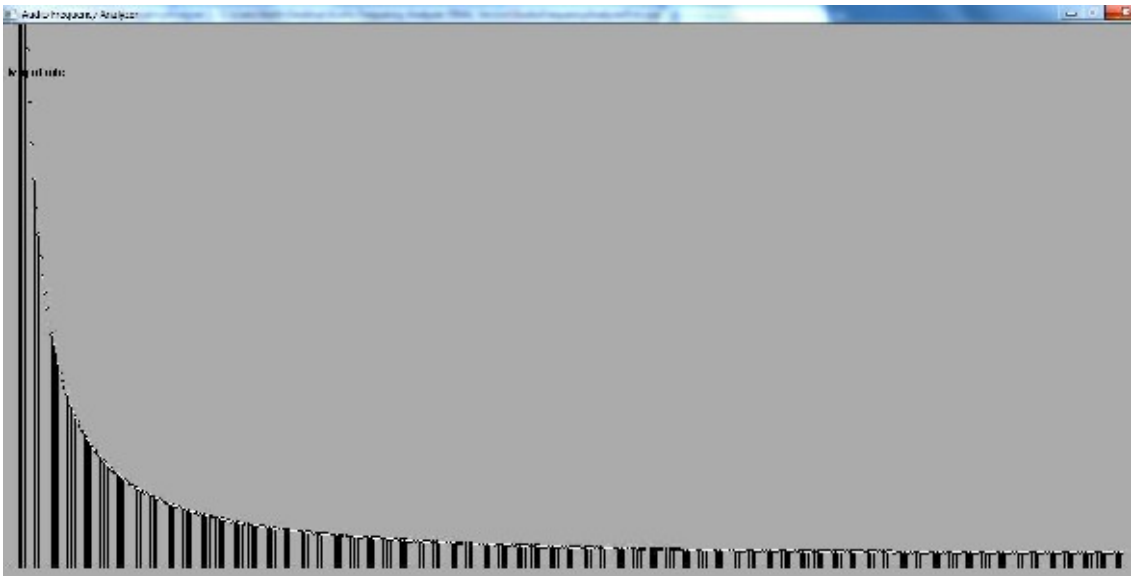


Figure 6.2.4 1024-point frequency spectrum using the DFT test code

6.3. Constant Input

Another test that we did is to generate a constant input and see if it produces an impulse function in the display. Indeed, a spike at point 0 was produced, as can be seen from the figure below.

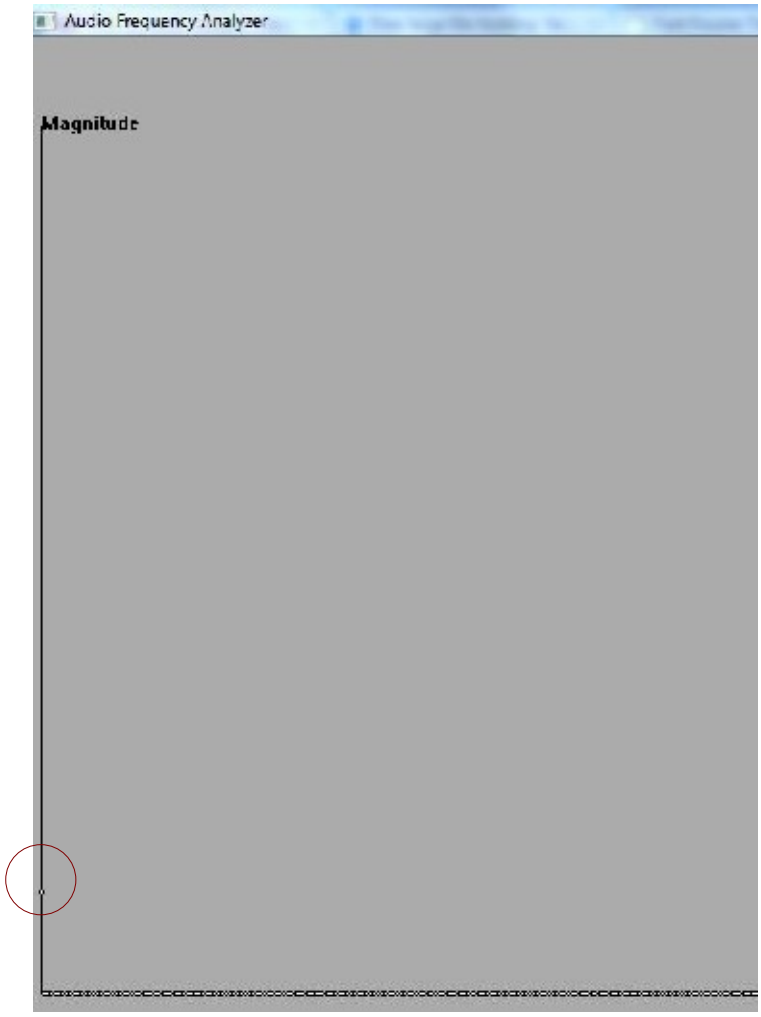


Figure 6.3.1 Impulse function in frequency spectrum

6.4. Sinusoidal Waves with Different Frequencies

Using the software NCH Tone Generator, we generated audible sinusoidal waves with different frequencies. The frequencies generated were 500 Hz, 1000Hz, 1500Hz, 2000Hz,

2500Hz, 3000Hz, 3500Hz, and 4000Hz. Each one was saved as a .wav file and played in a silent room while the audio frequency analyzer was running. Also, the volume of the playback was adjusted during each tone playback such that noise is as less noticeable as possible. This experiment was done so that the analyzer may be checked if it displays correct peaks at different frequencies. The results of the tests are as follows:

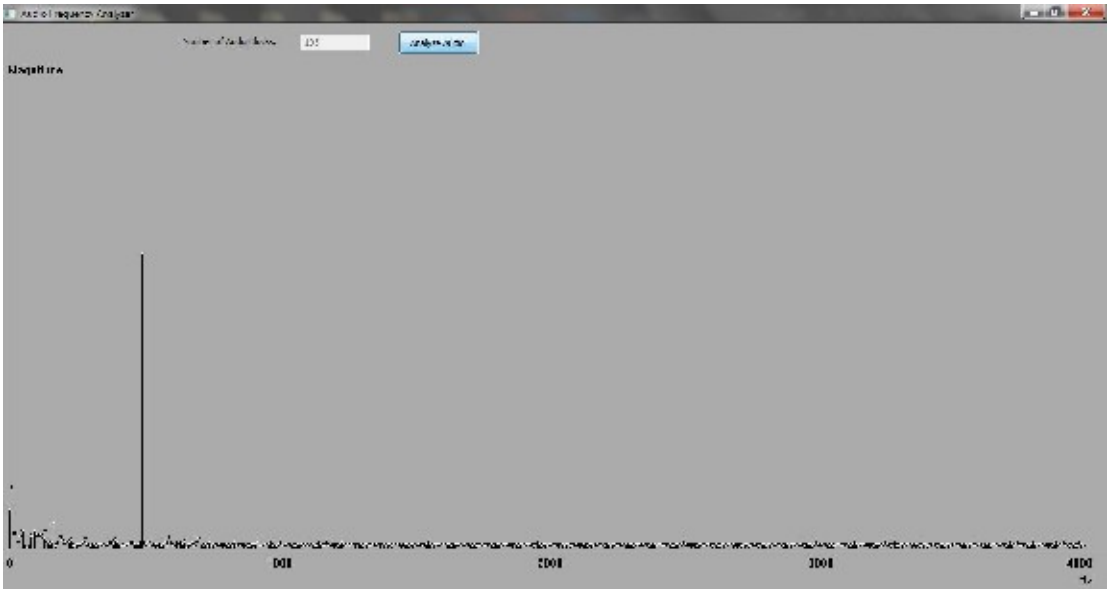


Fig. 6.4.1 Analysis of a 500-Hz sinusoidal waveform.

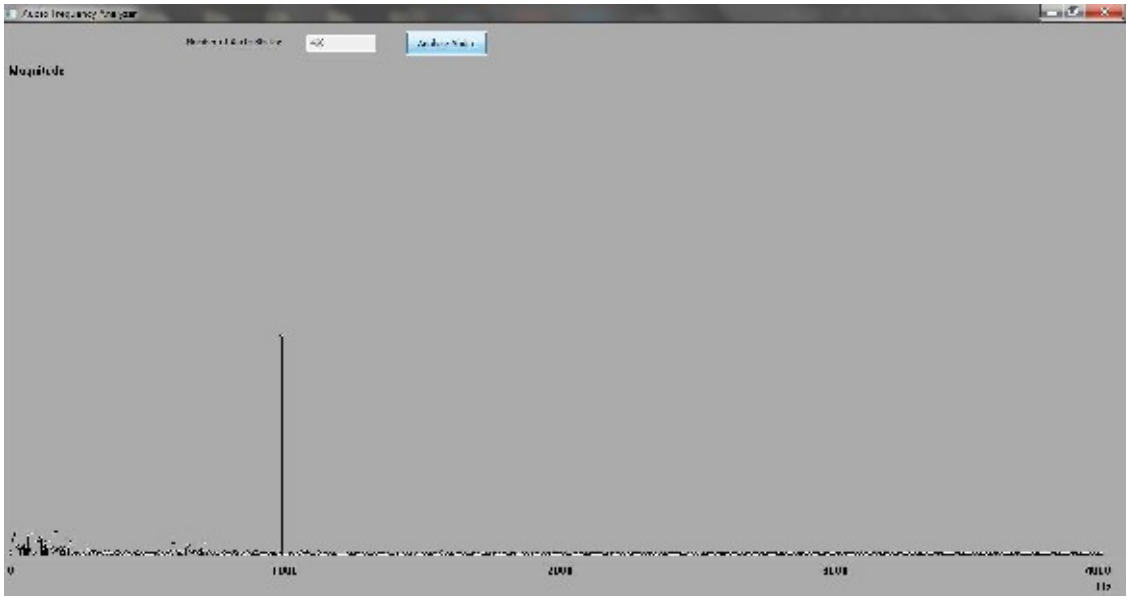


Fig. 6.4.2 Analysis of a 1000-Hz sinusoidal waveform.

Fig. 6.4.3 Analysis of a 1500-Hz sinusoidal waveform.

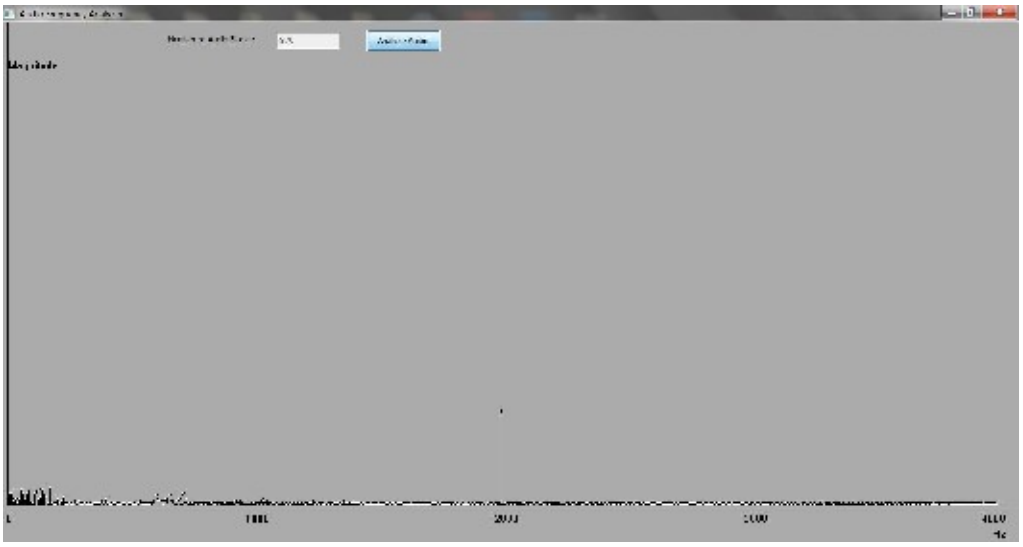


Fig. 6.4.4 Analysis of a 2000-Hz sinusoidal waveform.

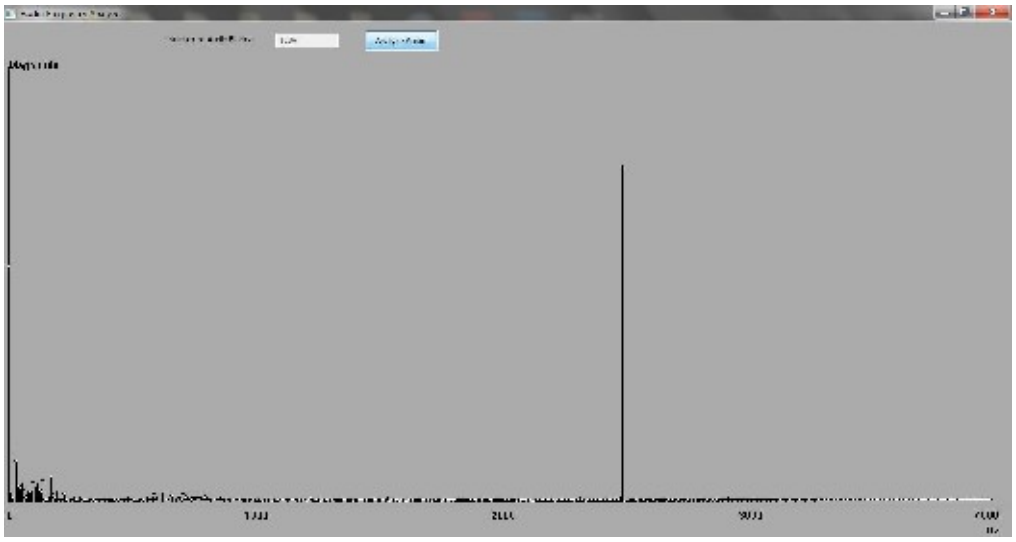


Fig. 6.4.5 Analysis of a 2500-Hz sinusoidal waveform.

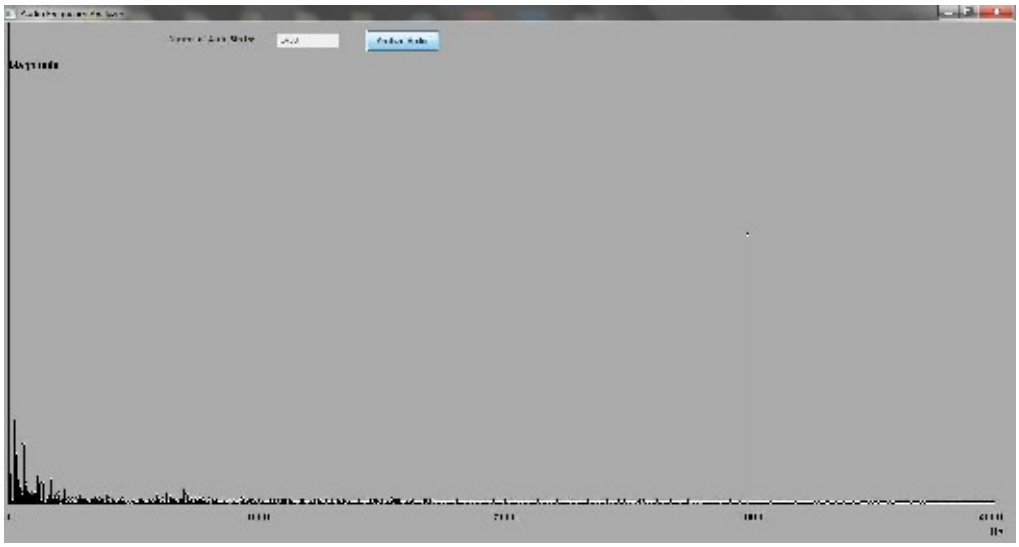


Fig. 6.4.6 Analysis of a 3000-Hz sinusoidal waveform.

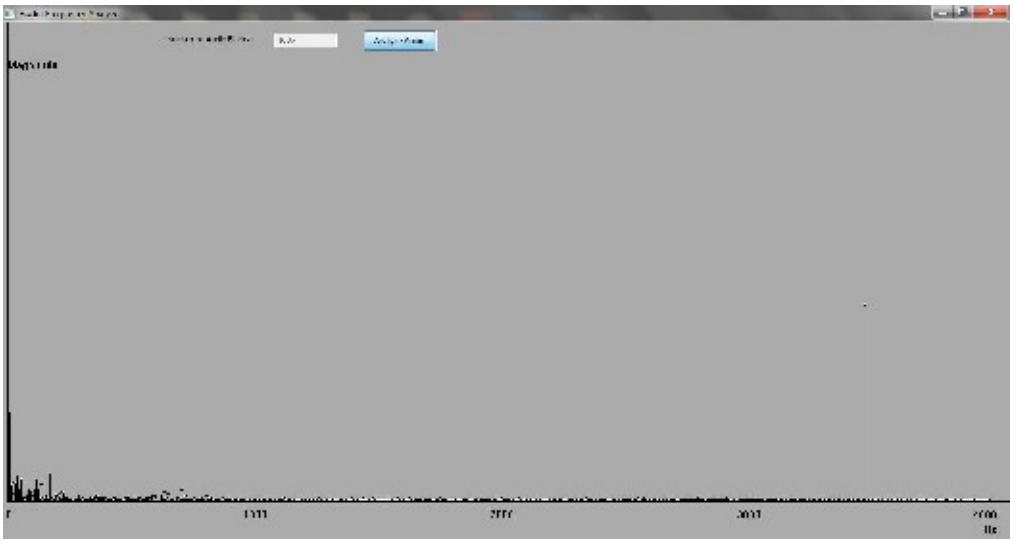


Fig. 6.4.7 Analysis of a 3500-Hz sinusoidal waveform.

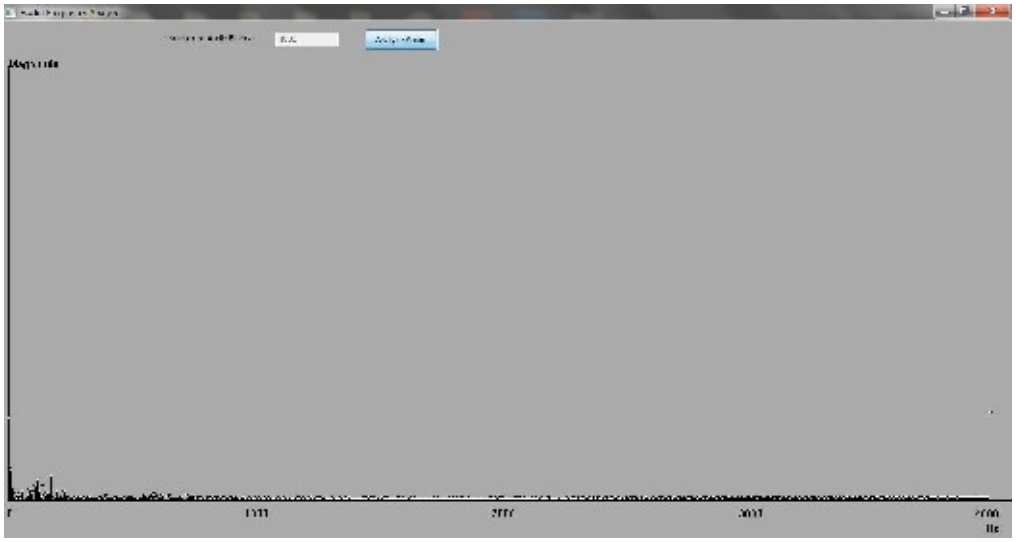


Fig. 6.4.8 Analysis of a 4000-Hz sinusoidal waveform.

6.5 Monophonic, Polyphonic, and True Tones

To further test the accuracy of the audio frequency analyzer, we varied the input audio from monophonic tones to polyphonic tones and finally to true tones. The monophonic tone was generated by playing a recorded Nokia 3310 ringing tone. The polyphonic tone was generated by a midi audio file which was played from a Sony Ericsson R300 cellular phone. And finally, the true tone was generated by an mp3 file of the song “Imagine” by the Glee Cast. We used these three types of tones because their frequency spectrum should noticeably vary from one type to another. A monophonic tone is expected to have the least frequency components since it only uses one note or one instrument at a time. In contrast, a polyphonic tone can play 40 notes at a time so it is expected to produce more frequency components than the monophonic tone. Then, the true tone is expected to display the most frequency components since it is a recording of actual music or sound. (cellutips.com, 2009) These expectations were more or less achieved and the proof are as follows:

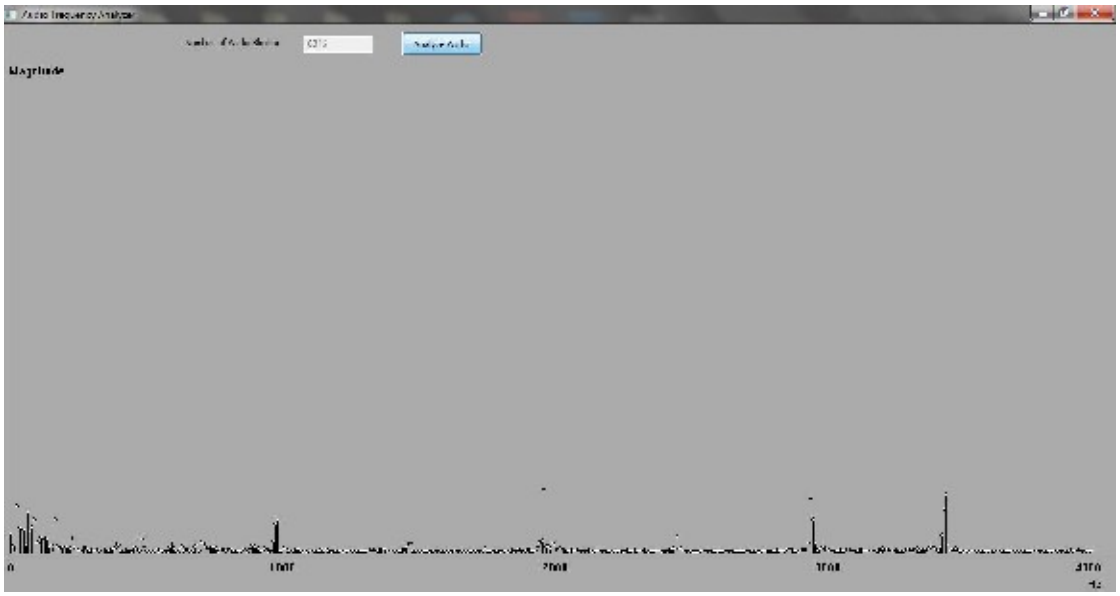


Fig. 6.5.1 Analysis of a monophonic ringing tone.

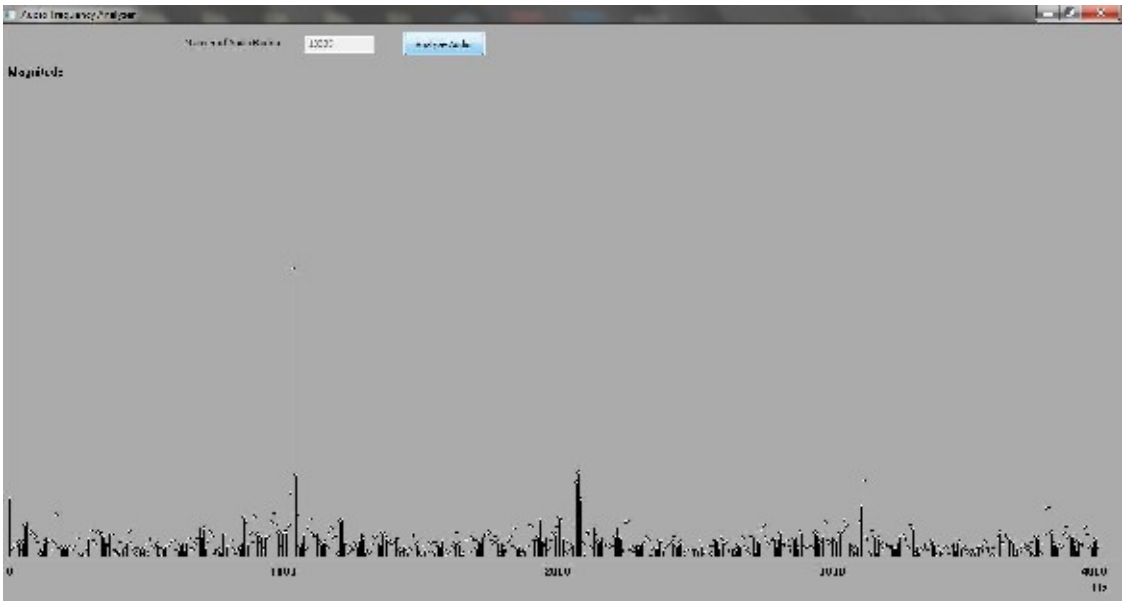


Fig. 6.5.2 Analysis of a polyphonic ringing tone.

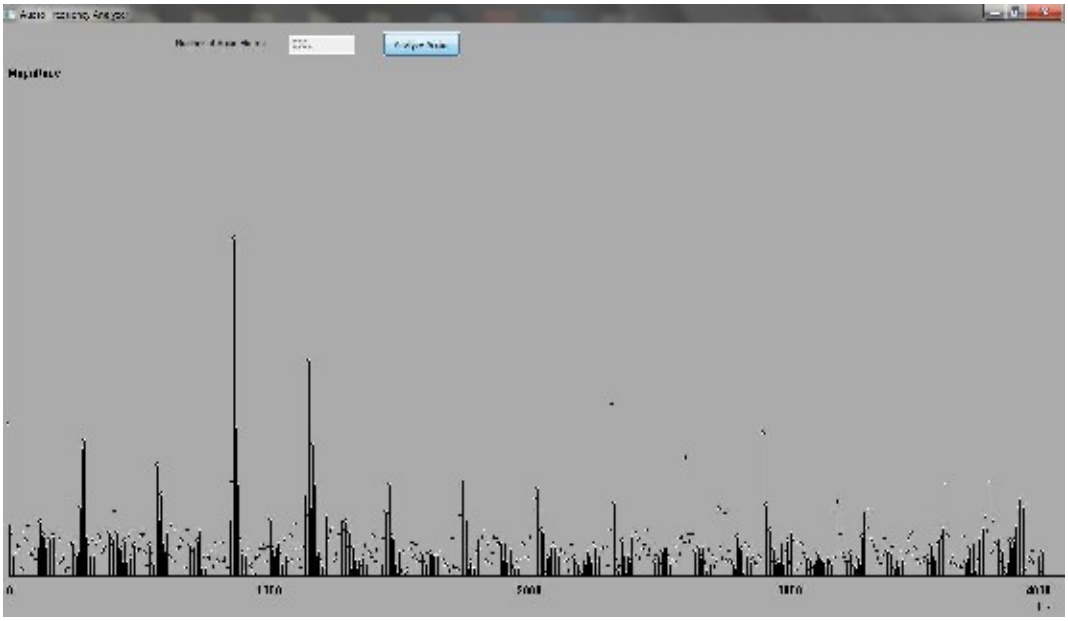


Fig. 6.5.3 Analysis of a true tone.

6.6. Theoretical vs Actual Audio Blocks Processed

To check whether the audio processing of the analyzer is real-time or not, we compared the actual number of audio blocks recorded using a timer and the indicator of audio blocks processed at the top of the GUI. Note that the number of audio blocks that should be recorded if the recorder samples at 8000 samples per second with the audio block size set to 1024 samples is theoretically about 7.8 blocks. So in 1 minute, the number of blocks that should be recorded is 468. In our experiment, the actual number of blocks processed in 59.67sec is 344. In 59.67sec, the theoretical number of audio blocks is about 465.4. This means that about 121 blocks were missed by the program. Conversely, the program is able to process about 73.9% per minute. And it only misses about 26.1% of the total theoretical audio blocks per minute. The screen shot of the test is shown in Figure 6.6.1.

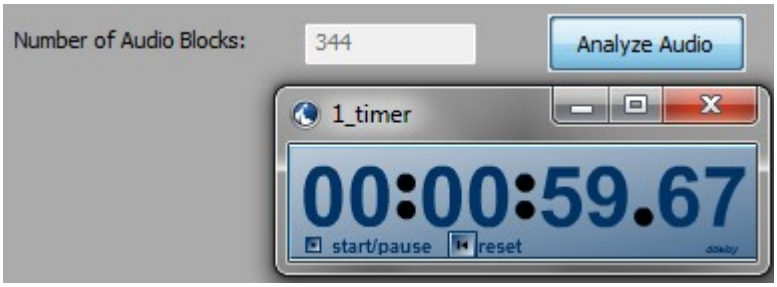


Fig. 6.6.1. Screen shot of the timer and the number of audio blocks processed.

6.7 Recorder Testing

The recorder part of this project was tested using the play and save functions of the XMCS player/recorder. Before implementing the recorder on the GUI, we first tested the function in a console application. At first, we created a program that records and then plays the recorded audio right away. A keyboard hit will trigger the start and end of the recording process. The next keyboard hit will play the recorded audio. This test makes sure that recording does take place with the record function.

Since the final output of this project is a windows application, we excluded the

keyboard hit part which is only applicable to console applications. To stop the recording process, we replaced the keyboard hit function with a loop that will stop when the audio block size is full. The audio block size was also set to 1024 samples or 2048 bytes for 16bits per sample. After doing so, we tested the program by playing the audio after recording. We expected short audio lengths to be played depending on the audio block size that we set. This test was also successful.

Finally, we had to access the audio blocks that were recorded so we used the save function since this is the part of the original code that directly accesses the audio samples in the memory buffers. First, we created a program that would display the audio blocks that were recorded using printf() or cout. The audio blocks that were displayed were then compared with the ones in the files generated by the save function that was also included in the code right after recording. This gave us the assurance that we are indeed accessing the correct audio samples from the buffers.

7. Results and Recommendations

A simple audio frequency analyzer with its basic function has been successfully implemented. Its main features are as follows:

- support for 16-bit audio
- flexible duration for recording

To further improve the project, it is recommended that the following features be added:

- use of a range of sampling rates for the XMCS files;
- pause-and-resume, and stop operation during the recording by using threads;
- and better resolution of the graph of the frequency spectrum and added options that will allow the user to record and graph in different sampling rates.

Appendix

Source Code

A.1. Source Files

A.1.1. audiofrequencyanalyzerfrm.cpp

```
///-----------------------------------------------------------------------
///
/// @file      audiofrequencyanalyzerfrm.cpp
/// @author    Gerrald Mateo, Keith Pulmano, Mae Villanueva
/// Created:   9/18/2010 9:49:04 PM
/// @section   DESCRIPTION
///            audiofrequencyanalyzerfrm class implementation
///
///-----------------------------------------------------------------------

#include "audiofrequencyanalyzerfrm.h"
#include "xmcs.h"
#include "utility.h"
#include "fourier.h"

//-----------------------------------------------------------------------
// audiofrequencyanalyzerfrm
//-----------------------------------------------------------------------

////Event Table Start
BEGIN_EVENT_TABLE(audiofrequencyanalyzerfrm,wxFrame)
      EVT_CLOSE(audiofrequencyanalyzerfrm::OnClose)

EVT_BUTTON(ID_ANALYZEAUDIO,audiofrequencyanalyzerfrm::WxButton1Click)
END_EVENT_TABLE()
////Event Table End

audiofrequencyanalyzerfrm::audiofrequencyanalyzerfrm(wxWindow *parent,
      wxWindowID id, const wxString &title, const wxPoint &position,
      const wxSize& size, long style)
      : wxFrame(parent, id, title, position, size, style)
{
      graph = new wxClientDC(this);
      CreateGUIControls();
}

audiofrequencyanalyzerfrm::~audiofrequencyanalyzerfrm()
{
}

void audiofrequencyanalyzerfrm::CreateGUIControls()
{
```

```
	////GUI Items Creation Start

	numberOfAudioBlocks = new wxStaticText(this,
		ID_NUMBEROFAUDIOBLOCKS, wxT("Number of Audio Blocks:"),
		wxPoint(222, 16), wxDefaultSize, 0,
		wxT("numberOfAudioBlocks"));

	countAudioBlocks = new wxTextCtrl(this, ID_AUDIOBLOCKS, wxT(""),
		wxPoint(366, 14), wxSize(87, 22), 0, wxDefaultValidator,
		wxT("countAudioBlocks"));
	countAudioBlocks->Enable(false);

	analyzeAudio = new wxButton(this, ID_ANALYZEAUDIO, wxT("Analyze
		Audio"), wxPoint(488, 10), wxSize(99, 29), 0,
		wxDefaultValidator, wxT("analyzeAudio"));

	SetTitle(wxT("Audio Frequency Analyzer"));
	SetIcon(wxNullIcon);
	SetSize(8,8,783,240);
	Center();

	////GUI Items Creation End
}

void audiofrequencyanalyzerfrm::OnClose(wxCloseEvent& event)
{
	Destroy();
}

void audiofrequencyanalyzerfrm::WxButton1Click(wxCommandEvent& event)
{
	int ctr = 0;
	while(1)
	{
		//record audio from soundcard
		audiofrequencyanalyzerfrm::audio.Record();

		//for FFT
		int j=0;
		Fourier compute;
		int data[1024];
		double out[1024];
		int np=1024;

		//transfer audio blocks from linked list to array
		for (int i=0; i<np; i++) data[i]=0;
		audio.dataList.clear();
		audiofrequencyanalyzerfrm::audio.AccessBlocks();
			for(audio.it=audio.dataList.begin(); audio.it!=
				audio.dataList.end(); audio.it++)
			{
				if (j<np)
				{
				data[j]=*audio.it;
```

```
                        j++;
                        }
                  }

            //take the FFT of arrays
            compute.set(data, np);
            compute.FFT_function();
            compute.magnitude(out);

            graph->Clear();
            graph->DrawText("Magnitude",5, 50);
            graph->DrawText("Hz", 1330, 680);
            graph->DrawText("0", 3, 660);
            graph->DrawText("1000", 325, 660);
            graph->DrawText("2000", 660, 660);
            graph->DrawText("3000", 995, 660);
            graph->DrawText("4000", 1315, 660);
            graph->DrawLine(5, 650, 2000, 650);
            graph->DrawLine(5, 60, 5, 650);

            int x = 0;
            int y = 0;

            for (int i=0; i<513; i++)
            {
                  x = 5 + (i*2.6);
                  y = 650-(out[i]*2);
                  graph->DrawLine(x, 650, x, y);
                  graph->DrawCircle(x, y, 2);
            }
            ctr++;
            countAudioBlocks->Clear();
            *countAudioBlocks<<ctr;
      }
}
```

A.1.2. audiofrequencyanalyzerfrm.h

```
///-----------------------------------------------------------------
///
/// @file      audiofrequencyanalyzerfrm.h
/// @author    Gerrald Mateo, Keith Pulmano, Mae Villanueva
/// Created:   9/18/2010 9:49:04 PM
/// @section   DESCRIPTION
///            audiofrequencyanalyzerfrm class declaration
///
///-----------------------------------------------------------------

#ifndef __AUDIOFREQUENCYANALYZERFRM_H__
#define __AUDIOFREQUENCYANALYZERFRM_H__

#ifdef __BORLANDC__
      #pragma hdrstop
#endif

#ifndef WX_PRECOMP
      #include <wx/wx.h>
      #include <wx/frame.h>
#else
      #include <wx/wxprec.h>
#endif
    #include <wx/dcclient.h>
    #include <wx/colour.h>
    #include "xmcs.h"
    #include "utility.h"
    #include <complex>

////Header Include Start
#include <wx/stattext.h>
#include <wx/textctrl.h>
#include <wx/button.h>
////Header Include End

////Dialog Style Start
#undef audiofrequencyanalyzerfrm_STYLE
#define audiofrequencyanalyzerfrm_STYLE wxCAPTION | wxRESIZE_BORDER |
      wxSYSTEM_MENU | wxMINIMIZE_BOX | wxMAXIMIZE_BOX | wxCLOSE_BOX
////Dialog Style End

class audiofrequencyanalyzerfrm : public wxFrame
{
      private:
            DECLARE_EVENT_TABLE();

      public:
            audiofrequencyanalyzerfrm(wxWindow *parent, wxWindowID id =
                  1, const wxString &title = wxT("Audio Frequency
                  Analyzer"), const wxPoint& pos = wxDefaultPosition,
                  const wxSize& size = wxDefaultSize, long style =
                  audiofrequencyanalyzerfrm_STYLE);
```

```
            virtual ~audiofrequencyanalyzerfrm();
            XMCS audio;
            void WxButton1Click(wxCommandEvent& event);

      private:
            ////GUI Control Declaration Start
            wxClientDC *graph;
            wxTextCtrl *countAudioBlocks;
            wxButton *analyzeAudio;
            wxStaticText *numberOfAudioBlocks;

      private:
            enum
            {
                  ////GUI Enum Control ID Start
                  ID_NUMBEROFAUDIOBLOCKS = 1009,
                  ID_AUDIOBLOCKS = 1008,
                  ID_ANALYZEAUDIO = 1006,
                  ////GUI Enum Control ID End
                  ID_DUMMY_VALUE_
            };

      private:
            void OnClose(wxCloseEvent& event);
            void CreateGUIControls();
};

#endif
```

A.1.3. audiofrequencyanalyzerapp.cpp

```
//---------------------------------------------------------------------------
-----
//
// Name:        audiofrequencyanalyzerfrm.cpp
// Author:      Gerrald Mateo, Keith Pulmano, Mae Villanueva
// Created:     9/18/2010 9:49:03 PM
// Description:
//
//---------------------------------------------------------------------------
-----

#include "audiofrequencyanalyzerapp.h"
#include "audiofrequencyanalyzerfrm.h"

IMPLEMENT_APP(audiofrequencyanalyzerfrmapp)

bool audiofrequencyanalyzerfrmapp::OnInit()
{
      audiofrequencyanalyzerfrm* frame = new
            audiofrequencyanalyzerfrm(NULL);
```

```
	SetTopWindow(frame);
	frame->Show();
	return true;
}

int audiofrequencyanalyzerfrmapp::OnExit()
{
	return 0;
}
```

A.1.4. audiofrequencyanalyzerapp.h

```
//-------------------------------------------------------------------------
-----
//
// Name:        audiofrequencyanalyzerapp.h
// Author:      Gerrald Mateo, Keith Pulmano, Mae Villanueva
// Created:     9/18/2010 9:49:03 PM
// Description:
//
//-------------------------------------------------------------------------
-----

#ifndef __AUDIOFREQUENCYANALYZERFRMApp_h__
#define __AUDIOFREQUENCYANALYZERFRMApp_h__

#ifdef __BORLANDC__
	#pragma hdrstop
#endif

#ifndef WX_PRECOMP
	#include <wx/wx.h>
#else
	#include <wx/wxprec.h>
#endif

class audiofrequencyanalyzerfrmapp : public wxApp
{
	public:
		bool OnInit();
		int OnExit();
};

#endif
```

A.1.5. xmcs.cpp

```
#include "xmcs.h"
#include <stdio.h>
#include<iostream>
#include <complex>
using namespace std;

/*********FUNCTION DECLARATIONS**********/
//void bit_reverse_function(complex<double> *a, int length);
//void FFT_function(complex<double>*a, int length);
/*****************************************************************/
/*        XMCS     CLASS     CONSTRUCTORS     AND      DESTRUCTOR     */
/*****************************************************************/

XMCS::XMCS()
{
      // audio format
      wfx.nChannels       = 1;
      wfx.wBitsPerSample  = 16;
      wfx.nSamplesPerSec  = 8000;
      wfx.cbSize          = 0;
      wfx.wFormatTag      = WAVE_FORMAT_PCM;
      wfx.nBlockAlign     = (wfx.wBitsPerSample * wfx.nChannels) >> 3;
      wfx.nAvgBytesPerSec = wfx.nBlockAlign * wfx.nSamplesPerSec;

      // others
      radix               = 2;
      maxBufferSize       = 0;
      lastBufferSize      = 0;
}

XMCS::XMCS(XMCS& xmcs)
{
      wfx             = xmcs.wfx;
      radix           = xmcs.radix;
      maxBufferSize   = xmcs.maxBufferSize;
      lastBufferSize  = xmcs.lastBufferSize;

      int  block_size = maxBufferSize;
      char *buffer;

      list<char*>::iterator bli, ref;
      for (bli = xmcs.audioBuffer.begin() ; bli !=
            xmcs.audioBuffer.end(); bli++)
      {
            ref = bli;
            if (++ref == xmcs.audioBuffer.end())
                  block_size = lastBufferSize;
            buffer = new char[block_size];
            memcpy(buffer, *bli, block_size);
            audioBuffer.insert(audioBuffer.end(), buffer);
      }
}
```

```
XMCS::~XMCS()
{
    Unload();
}

/*************************************************************************/
/*           XMCS        CLASS       ACCESSOR        FUNCTIONS        */
/*************************************************************************/

bool XMCS::SetChannels(int val)
{
      if (!(val == 1 || val == 2))
      return false;

      wfx.nChannels       = val;
      wfx.nBlockAlign     = (wfx.wBitsPerSample * wfx.nChannels) >> 3;
      wfx.nAvgBytesPerSec = wfx.nBlockAlign * wfx.nSamplesPerSec;
      return true;
}

bool XMCS::SetBits(int val)
{
      if (!(val == 8 || val == 16))
            return false;

      wfx.wBitsPerSample  = val;
      wfx.nBlockAlign     = (wfx.wBitsPerSample * wfx.nChannels) >> 3;
      wfx.nAvgBytesPerSec = wfx.nBlockAlign * wfx.nSamplesPerSec;
      return true;
}

bool XMCS::SetSamplesPerSecond(int val)
{
      //if (!(val == 8000 || val == 11025 || val == 22050 || val ==
            44100))
      if (val < 8000 || val > 44100)
            return false;

      wfx.nSamplesPerSec  = val;
      //wfx.nBlockAlign     = (wfx.wBitsPerSample * wfx.nChannels) >> 3;
      wfx.nAvgBytesPerSec = wfx.nBlockAlign * wfx.nSamplesPerSec;
      return true;
}

bool XMCS::SetFileRadix(int val)
{
      if (!(val == 2 || val == 16))
            return false;

      radix = val;
      return true;
}
```

```
int XMCS::GetChannels()
{
      return wfx.nChannels;
}

int XMCS::GetBits()
{
      return wfx.wBitsPerSample;
}

int XMCS::GetSamplesPerSecond()
{
      return wfx.nSamplesPerSec;
}
int XMCS::GetFileRadix()
{
      return radix;
}


/*************************************************************************/
/*        XMCS       CLASS         FILE       I/O      OPERATIONS     */
/*************************************************************************/

bool XMCS::ReadFileHeader(ifstream& fin)
{
      string keyword;
      string info;

      int t_channels, t_bits, t_radix, t_samplesPerSec;

      fin>>keyword;
      if (!(keyword == "channels"))
            return false;
      fin>>info;
      if (Utility::IsDecimal(info))
      {
            t_channels = atoi(info.c_str());
            if (!(t_channels == 1 || t_channels == 2))
                  return false;
      }
      else
            return false;

      fin>>keyword;
      if (!(keyword == "bits"))
            return false;
      fin>>info;
      if (Utility::IsDecimal(info))
      {
            t_bits = atoi(info.c_str());
            if (!(t_bits == 8 || t_bits == 16))
                  return false;
      }
```

```
	else
		return false;

	fin>>keyword;
	if (!(keyword == "radix"))
		return false;
	fin>>info;
	if (Utility::IsDecimal(info))
	{
		t_radix = atoi(info.c_str());
		if (!(t_radix == 2 || t_radix == 16))
			return false;
	}

	else
		return false;
	fin>>keyword;
	if (!(keyword == "format"))
		return false;

	fin>>info;
	if (Utility::IsDecimal(info))
	{
		t_samplesPerSec = atoi(info.c_str());
		//if (!(t_samplesPerSec == 8000 || t_samplesPerSec == 11025
			|| t_samplesPerSec == 22050 || t_samplesPerSec ==
			44100))
		if (t_samplesPerSec < 8000 || t_samplesPerSec > 44100)
			return false;
	}
	else
		return false;

	fin>>keyword;
	if (!(keyword == "formatend"))
		return false;

	SetChannels(t_channels);
	SetBits(t_bits);
	SetSamplesPerSecond(t_samplesPerSec);
	SetFileRadix(t_radix);
	return true;
}

bool XMCS::WriteFileHeader(ofstream& fout)
{
	int channels		= GetChannels();
	int bits		= GetBits();
	int samplesPerSec	= GetSamplesPerSecond();

	fout<<"channels "<<channels<<"\nbits "<<bits<<"\nradix "<<radix
		<<"\nformat "<<samplesPerSec<<" formatend\n";
```

```
if (fout.fail())
            return false;
      return true;
}

int XMCS::ReadBlockToBuffer(ifstream& fin, int bufferSize, char* buffer)
{
      int bits = GetBits();

      if (bufferSize%2 || bufferSize < 0)
            return -1;

      int readBytes = 0;

      if (fin.eof())
      {
            readBytes = 0;
            return readBytes;
      }

      switch (bits)
      {
            case 8:
            {
                  for (int i = 0; i < bufferSize; i++)
                  {
                        buffer[i] = GetNextSample(fin);
                        readBytes = fin.eof() ? (i + 1) : bufferSize;
                        if (fin.eof())
                              break;
                  }
            }
            break;

            case 16:
            {
                  for (int i = 0; i < bufferSize; i+=2)
                  {

                        short int nextSample = GetNextSample(fin);
                        buffer[i+1] = nextSample & 0xFF;
                        buffer[i] = (nextSample & 0xFF00) >> 8;
                        readBytes = fin.eof() ? (i + 1) : bufferSize;
                        if (fin.eof())
                              break;
                  }
            }
            break;

            default:
            return 0;
      }
      return readBytes;
}
```

```
int XMCS::WriteBlockFromBuffer(ofstream& fout, int bufferSize, const
      char* buffer)
{
      int channels = GetChannels();
      int bits = GetBits();

      if (/*bufferSize%2 ||*/ bufferSize < 0)
            return 0;
      const int samplesPerLine = channels * 2;
      fout<<"\n";
      switch (bits)
      {
            case 8:
            {
                  for (int i = 0; i < bufferSize; i++)
                  {
                        if (!(i % samplesPerLine))
                              fout<<"\n";

                        if (radix == 2)
                              fout<<Utility::Int16ToBinaryString(buffer[
                                    i],8)<<" ";
                        else if (radix == 16)
                              fout<<Utility::Int16ToHexString(buffer[i],
                                    8)<<" ";
                  }
            }
            break;

            case 16:
            {
                  short int num16;
                  for (int i = 0; i < bufferSize; i+=2)
                  {
                        num16 = ((unsigned char)buffer[i] << 8) ^
                              ((unsigned char)buffer[i+1]);

                        if (!((i/2) % samplesPerLine))
                              fout<<"\n";

                        if (radix == 2)
                              fout<<num16<<endl;
                        //Utility::Int16ToBinaryString(num16,16)<<" ";
                        else if (radix == 16)
                              fout<<Utility::Int16ToHexString(num16,16)<
                                    <" ";
                  }
            }
            break;
            default:
            return 0;
      }
            return bufferSize;
}
```

```
short int XMCS::GetNextSample(ifstream& fin)
{
      int bits = GetBits();

      char    temp;
      string  sample;

      sample.clear();

      switch(radix)
      {
            case 2:

            for (int i = 0; i < bits; i++)
            {
                  temp = GetNextNonWhiteSpace(fin);

                 // an unexpected character means start of comment upto
                        end of line
                  while (!Utility::IsBinaryDigit(temp))
                  {
                        DiscardUpToEol(fin);
                        temp = GetNextNonWhiteSpace(fin);
                        if (fin.eof())
                              break;
                  }
                  sample += temp;
            }
            return Utility::BinaryStringToInt16(sample);
            break;

            case 16:

            for (int i = 0; i < bits/4; i++)
            {
                  temp = GetNextNonWhiteSpace(fin);

                 // an unexpected character means start of comment upto
                        end of line
                  while (!isxdigit(temp))
                  {
                        DiscardUpToEol(fin);
                        temp = GetNextNonWhiteSpace(fin);
                        if (fin.eof())
                              break;
                  }

                  sample += temp;
            }
            return Utility::HexStringToInt16(sample);
            break;
    }
    return 0;
}
```

```
char XMCS::GetNextNonWhiteSpace(ifstream& fin)
{
	char digit;

	do
	{
		if (lineCharsRead == 256)
		{
			DiscardUpToEol(fin);
		}
		digit = fin.get();
		lineCharsRead = (digit == '\n') ? 0 : (lineCharsRead + 1);

	} while (isspace(digit) && !fin.eof());

	if (fin.eof())
		return 0;

	return digit;
}

void XMCS::DiscardUpToEol(ifstream& fin)
{
	char c;
	do
	{
		if (fin.eof())
			break;
		c = fin.get();
	} while (c != '\n');
	lineCharsRead = 0;
}

/*********************************************************************/
/*           XMCS      CLASS      SPECIAL       FUNCTIONS             */
/*********************************************************************/

bool XMCS::Load(string filename)
{
	const int BLOCK_SIZE = 1024;
	char *tempBuffer;
	char *buffer;

	ifstream fin;
	fin.open(filename.c_str());
	if (fin.fail())
		return false;
	if (!ReadFileHeader(fin))
		return false;

	Unload();

	tempBuffer = new char[BLOCK_SIZE];
	lineCharsRead = 0;
```

```
	do
	{
		buffer = new char[BLOCK_SIZE];
		lastBufferSize = ReadBlockToBuffer(fin, BLOCK_SIZE,
			tempBuffer);
		memcpy(buffer, tempBuffer, BLOCK_SIZE);
		audioBuffer.insert(audioBuffer.end(), buffer);
	} while (lastBufferSize == BLOCK_SIZE);

	maxBufferSize = BLOCK_SIZE;

	delete [] tempBuffer;
	fin.close();
}


bool XMCS::Unload()
{
	if (!audioBuffer.size())
		return false;

	int i = audioBuffer.size();
	int j = 0;

	while (audioBuffer.size() && maxBufferSize)
	{
		if (audioBuffer.size() != 1 || lastBufferSize)
			delete [] (*audioBuffer.begin());
		audioBuffer.erase(audioBuffer.begin());
	}
	lastBufferSize = maxBufferSize = 0;

	return true;
}

bool XMCS::Save(string filename)
{
	int block_size = maxBufferSize;
	ofstream fout;
	fout.open(filename.c_str());
	if (fout.fail())
		return false;

	if (!WriteFileHeader(fout))
		return false;

	list<char*>::iterator bli, ref;
	for (bli = audioBuffer.begin(); bli != audioBuffer.end(); bli++)
	{
		ref = bli;
		if (++ref == audioBuffer.end())
			block_size = lastBufferSize;
		WriteBlockFromBuffer(fout, block_size, *bli);
	}
```

```
	fout.close();
	return true;
}

// accesses audioBuffer and transfers into integer list
void XMCS::AccessBlocks()
{
	int block_size = maxBufferSize;
	list<char*>::iterator bli, ref;
	for (bli = audioBuffer.begin(); bli != audioBuffer.end(); bli++)
	{
		ref = bli;
		if (++ref == audioBuffer.end())
		block_size = lastBufferSize;
		int bufferSize = block_size;
		const char* buffer = *bli;
		for (int i = 0; i < bufferSize; i+=2)
		{
			short int num16;
			num16 = ((unsigned char)buffer[i] << 8) ^ ((unsigned
				char)buffer[i+1]);
			dataList.push_back(num16);    //list that contains
				integer audio blocks
		}
	}
}

bool XMCS::Play()
{
	const int BLOCK_COUNT   = 10;   // number of audio blocks for
					    playback

	HWAVEOUT    hWaveOut;           // handle
	WAVEHDR     *waveBlocks;        // array of wave headers
	int         waveCurrentBlock;   // index of current block being
					    played
	int         block_size;         // size of block to be played
	char        *buffer;            // temporary buffer

	// INITIALIZATION AND MEMORY ALLOCATION

	block_size = maxBufferSize;
	waveBlocks = new WAVEHDR[BLOCK_COUNT];
	for (int i = 0; i < BLOCK_COUNT; i++)
	{
		memset(&waveBlocks[i], 0, sizeof(WAVEHDR));
		waveBlocks[i].lpData = new char[block_size];
	}

	freeBlockCount  = BLOCK_COUNT;
	waveCurrentBlock    = 0;

	InitializeCriticalSection(&criticalSection);
```

```
// PLAYBACK

// open audio output device
      if(waveOutOpen(&hWaveOut, WAVE_MAPPER, &wfx,
            (DWORD_PTR)waveOutProc, (DWORD_PTR)this,
            CALLBACK_FUNCTION) != MMSYSERR_NOERROR)
      return false;

list<char*>::iterator bli = audioBuffer.begin();
list<char*>::iterator ref;

buffer  = new char[block_size];

// unless playback is stopped, play everything contained in the
      list
while(bli != audioBuffer.end())
{
      if (kbhit())
      {
            getch();
            waveOutReset(hWaveOut);
            break;
      }

      // copy data to be used to buffer
      memcpy(buffer, *bli, block_size);
      ref = bli++;

      // if the last element of the list, size may not be equal to
            the maximum
      if (++ref == audioBuffer.end())
            block_size = lastBufferSize;

      // unprepare header to be used for playback, in case not yet
            unprepared
      if(waveBlocks[waveCurrentBlock].dwFlags & WHDR_PREPARED)
            waveOutUnprepareHeader(hWaveOut,
                  &waveBlocks[waveCurrentBlock], sizeof(WAVEHDR));

      // specify buffer length and audio samples
            waveBlocks[waveCurrentBlock].dwBufferLength =
                  sizeof(char)*block_size;
            memcpy(waveBlocks[waveCurrentBlock].lpData, buffer,
                  sizeof(char)*block_size);

      // prepare header to be used for playback
            if (waveOutPrepareHeader(hWaveOut,
                  &waveBlocks[waveCurrentBlock],
                  sizeof(WAVEHDR)) != MMSYSERR_NOERROR)
                  return false;

      // play
            waveOutWrite(hWaveOut, &waveBlocks[waveCurrentBlock],
                  sizeof(WAVEHDR));
```

```
            // one less temporary buffer available
            EnterCriticalSection(&criticalSection);
            freeBlockCount--;
            LeaveCriticalSection(&criticalSection);

            // unless playback is stopped, wait until there's a free
                  audio block
            while(!freeBlockCount)
            {
                  if (kbhit())
                  {
                        getch();
                        waveOutReset(hWaveOut);
                        bli = audioBuffer.end();
                        break;
                  }
            }

            // current block is next block (imagine array is circular)
            waveCurrentBlock++;
            waveCurrentBlock %= BLOCK_COUNT;
      }

      // unless playback is stopped, wait for all blocks to complete
      while(freeBlockCount < BLOCK_COUNT)
      {
            if (kbhit())
            {
                  getch();
                  waveOutReset(hWaveOut);
                  break;
            }

      }

      // CLEAN-UP

      // unprepare any blocks that are still prepared
      for(int i = 0; i < freeBlockCount; i++)
            if(waveBlocks[i].dwFlags & WHDR_PREPARED)
                  waveOutUnprepareHeader(hWaveOut, &waveBlocks[i],
                        sizeof(WAVEHDR));

      DeleteCriticalSection(&criticalSection);
      waveOutClose(hWaveOut);

      delete [] buffer;

      for (int i = 0; i < BLOCK_COUNT; i++)
            delete [] waveBlocks[i].lpData;
      delete [] waveBlocks;

      return true;
}
```

```
bool XMCS::Record()
{
      Unload(); // clear list

      HWAVEIN hWaveIn[2];          // handles
      WAVEHDR waveHeader[2];       // wave headers
      bool    isFirst;             // 1 indicates that first buffer is
                                         being filled
      bool    isStopped;           // 1 indicates recording was stopped
                                         by a key press
      char    *buffer;             // temporary buffer
      int     bufferSize;          // recording buffer size
      int     currentSet;          // set of handle and wave header being
                                         used
      int     otherSet;            // set of handle and wave header being
                                         prepared
      int     lastByteCount;       // number of recorded bytes on the
                                         last buffer

      // INITIALIZATION

      isFirst     = true;
      isStopped   = false;
      maxBufferSize = bufferSize = 2048;
      currentSet  = 0;

      // RECORDING

      // open audio input device and allocate memory for recorded
            samples, do for 2 sets
      // also, set values of wave header fields
      for (int i = 0; i < 2; i++)
      {
            // open device
            if(waveInOpen(&hWaveIn[i], WAVE_MAPPER, &wfx, 0, 0,
                  CALLBACK_NULL) != MMSYSERR_NOERROR)
                  return false;

            // allocate memory
            waveHeader[i].lpData = new char[bufferSize];

            // initialize
            waveHeader[i].dwBufferLength   = bufferSize;
            waveHeader[i].dwFlags          = 0;
            waveHeader[i].dwLoops          = 0;
      }

      // prepare a wave header, for the current set to be used for
            recording
      if (waveInPrepareHeader(hWaveIn[currentSet],
            &waveHeader[currentSet], sizeof(WAVEHDR)) !=
            MMSYSERR_NOERROR)
            goto failed_cleanup;
```

```
// add buffer, for the current set to be used for recording
if ((waveInAddBuffer(hWaveIn[currentSet], &waveHeader[currentSet],
      sizeof(WAVEHDR))) != MMSYSERR_NOERROR)
      goto failed_cleanup;

do
{
      // start recording
      if (waveInStart(hWaveIn[currentSet]) != MMSYSERR_NOERROR)
            goto failed_cleanup;


      buffer = new char[bufferSize];  // for buffer currently
            being recorded
      otherSet = 1 - currentSet; // (currentSet+1)%2;

      // for the first iteration (only), statements inside are not
            executed
      if (!isFirst)
      {
            memcpy(buffer,(LPSTR)(waveHeader[otherSet].lpData),
                  waveHeader[otherSet].dwBytesRecorded); // LIST
                  implementation
            audioBuffer.insert(audioBuffer.end(), buffer);

      }


      // do clean up for the other set (before preparing for
            another recording, if any)
      waveInStop(hWaveIn[otherSet]);
      waveInUnprepareHeader(hWaveIn[otherSet],
            &waveHeader[otherSet], sizeof(WAVEHDR));

      // prepare a wave header, for the other set, to be used for
            the next recording
      if (waveInPrepareHeader(hWaveIn[otherSet],
            &waveHeader[otherSet], sizeof(WAVEHDR)) !=
            MMSYSERR_NOERROR)
            goto failed_cleanup;

      // add buffer, for the other set, to be used for the next
            recording
      if ((waveInAddBuffer(hWaveIn[otherSet],
            &waveHeader[otherSet], sizeof(WAVEHDR))) !=
            MMSYSERR_NOERROR)
            goto failed_cleanup;

      // unless recording is stopped, wait until recording buffer
            is full
      while(!(waveHeader[currentSet].dwFlags & WHDR_DONE))
      {
            if (lastBufferSize==2048)
            {
```

```
                        //getch();
                        waveInReset(hWaveIn[0]);
                        waveInReset(hWaveIn[1]);
                        isStopped = true;
                        break;
                  }

            }


            // take note of number of bytes actually recorded
            lastBufferSize = lastByteCount =
                  waveHeader[currentSet].dwBytesRecorded;

            //cout<<lastBufferSize<<endl;
            // switch to the other set
            currentSet = 1 - currentSet; //(currentSet + 1) % 2;

            isFirst = false;

      }
      while (!isStopped);

      // add last recorded buffer to the list

      buffer = new char[bufferSize];
      otherSet = 1 - currentSet; //(currentSet+1)%2;
      memcpy(buffer,(LPSTR)(waveHeader[otherSet].lpData),
            waveHeader[otherSet].dwBytesRecorded);
      audioBuffer.insert(audioBuffer.end(), buffer);

      // CLEAN-UP

      for (int i = 0; i < 2; i++)
      {
            waveInStop(hWaveIn[i]);
            waveInUnprepareHeader(hWaveIn[i], &waveHeader[i],
                  sizeof(WAVEHDR));
            delete [] waveHeader[i].lpData;
            waveInClose(hWaveIn[i]);
      }

      return true;
      failed_cleanup:
      for (int i = 0; i < 2; i++)
      {
            waveInStop(hWaveIn[i]);
            waveInUnprepareHeader(hWaveIn[i], &waveHeader[i],
                  sizeof(WAVEHDR));
            delete [] waveHeader[i].lpData;
            waveInClose(hWaveIn[i]);
      }
      return false;
}
```

```
/*********************************************************************/
/*      CALLBACK    Procedure    for     XMCS     Audio    Playback   */
/*********************************************************************/

void   CALLBACK   waveOutProc(HWAVEOUT   hWaveOut,   UINT   uMsg,   DWORD
      dwInstance, DWORD dwParam1, DWORD dwParam2)
{
      XMCS* XMCS_ptr = (XMCS*) dwInstance;

      if(uMsg != WOM_DONE)
            return;

      EnterCriticalSection(&(XMCS_ptr->criticalSection));
            (XMCS_ptr->freeBlockCount)++;
      LeaveCriticalSection(&(XMCS_ptr->criticalSection));
}
```

A.1.6. xmcs.h

```
#ifndef _XMCS_H_
#define _XMCS_H_

#include "utility.h"
#include <windows.h>
#include <mmsystem.h>
#include <fstream>
#include <string>
#include <list>
#include <conio.h>

using std::string;
using std::ifstream;
using std::ofstream;
using std::list;

class XMCS
{
      private:

            /*
             * to store audio format
             */
             WAVEFORMATEX wfx;

            /*
             * list for audio samples, and buffer size indicators
             */
             list<char*> audioBuffer;
             int maxBufferSize;
             int lastBufferSize;

            /*
```

```
        * file radix, needed for reading and writing
        */
        int radix;

       /*
        * file input functions
        */
        char GetNextNonWhiteSpace(ifstream&);
        void DiscardUpToEol(ifstream&);
        short int GetNextSample(ifstream&);
        bool ReadFileHeader(ifstream&);
        int ReadBlockToBuffer(ifstream&, int, char*);

       /*
        * file output functions
        */
        bool WriteFileHeader(ofstream&);
        int WriteBlockFromBuffer(ofstream&, int, const char*);

       /*
        * others, shared by functions (like global variables)
        */
        short int lineCharsRead;
        CRITICAL_SECTION criticalSection;
        volatile int freeBlockCount;

public:

       /*
        * constructors, destructor
        */
        XMCS();
        XMCS(XMCS&);
        ~XMCS();
        list<int> dataList;
        list<int>::iterator it;
       /*
        * load audio samples from a file (filename) to the list
        *     (audioBuffer)
        */
        bool Load(string filename);

       /*
        * empty the list (audioBuffer)
        */
        bool Unload();

       /*
        * save audio samples in the list (audioBuffer) to a file
        *     (filename)
        */
        bool Save(string filename);

       /*
```

```
            * play audio samples in the list (audioBuffer)
            */
            bool Play();

           /*
            * record audio samples and store to the list (audioBuffer)
            */
            bool Record();

           /*
            * access audio blocks from audioBuffer and store to dataList
            */
            void AccessBlocks();

           /*
            * accessors
            */
            bool SetChannels(int);
            int  GetChannels();
            bool SetBits(int);
            int  GetBits();
            bool SetSamplesPerSecond(int);
            int  GetSamplesPerSecond();
            bool SetFileRadix(int);
            int  GetFileRadix();

           /*
            * callback procedure for playback
            */
            friend void CALLBACK waveOutProc(HWAVEOUT hWaveOut, UINT
                  uMsg, DWORD dwInstance, DWORD dwParam1, DWORD
                  dwParam2);
      };

#endif
```

A.1.7. utility.cpp

```
#include "utility.h"

char Utility::Symbol(int x)
{
	switch (x)
	{
		case 0:  return '0';
		case 1:  return '1';
		case 2:  return '2';
		case 3:  return '3';
		case 4:  return '4';
		case 5:  return '5';
		case 6:  return '6';
		case 7:  return '7';
		case 8:  return '8';
		case 9:  return '9';
		case 10: return 'A';
		case 11: return 'B';
		case 12: return 'C';
		case 13: return 'D';
		case 14: return 'E';
		case 15: return 'F';
		default: return 0;
	}
}

string Utility::Int16ToHexString(int num, int bits)
{
	if (bits != 8 && bits != 16)
		return "";

	char* hex = new char[bits/4 + 1];

	int mask[] = {0x00F0, 0x000F, 0xF000, 0x0F00};
	int shift[] = {4, 0, 12, 8};
	for (int i = 0; i < bits/4; i++)
	{
		hex[i] = Symbol((num & mask[i]) >> shift[i]);
	}

	hex[bits/4] = 0;
		string shex = (string)hex;
	delete [] hex;
	return shex;
}

string Utility::Int16ToBinaryString(int num, int bits)
{
	if (bits != 8 && bits != 16)
		return "";
```

```
	char* binary = new char[bits + 1];
	int mask[] = {0x0080, 0x0040, 0x0020, 0x0010, 0x0008, 0x0004,
		0x0002, 0x0001, 0x8000, 0x4000, 0x2000, 0x1000, 0x0800,
		0x0400, 0x0200, 0x0100};

	for (int i = 0; i < bits; i++)
	{
		binary[i] = Symbol(((num & mask[i]) == mask[i]) ? 1 : 0);
	}
	binary[bits] = 0;

	string sbinary = (string)binary;
	delete [] binary;
	return sbinary;
}


short int Utility::HexStringToInt16(string hex)
{
	if (!IsHex(hex))
		return 0;

	short int num = 0;
		for (int i = hex.length() - 1; i > 0; i-=2)
	{
		for (int j = i - 1; j <= i; j++)
		{
			int val;
			switch(toupper(hex[j]))
			{
				case '0' : val = 0;  break;
				case '1' : val = 1;  break;
				case '2' : val = 2;  break;
				case '3' : val = 3;  break;
				case '4' : val = 4;  break;
				case '5' : val = 5;  break;
				case '6' : val = 6;  break;
				case '7' : val = 7;  break;
				case '8' : val = 8;  break;
				case '9' : val = 9;  break;
				case 'A' : val = 10; break;
				case 'B' : val = 11; break;
				case 'C' : val = 12; break;
				case 'D' : val = 13; break;
				case 'E' : val = 14; break;
				case 'F' : val = 15; break;
				default  : val = 0;  break; // if unknown digit,
					assign 0
			}
		num = num*16 + val;
		}
	}
	return num;
}
```

```
short int Utility::BinaryStringToInt16(string bin)
{
      if (!IsBinary(bin))
            return 0;
      short int num = 0;
      int mark = (bin.length() > 8) ? (bin.length() - 8) : 0;

      for (int i = mark; i < bin.length(); i++)
      {
            num = num*2 + (bin[i] == '1' ? 1 : 0); // if not '1', then
                  assign 0
      }

      for (int i = 0; i < mark; i++)
      {
            num = num*2 + (bin[i] == '1' ? 1 : 0); // if not '1', then
                  assign 0
      }
      return num;
}

bool Utility::IsBinaryDigit(char digit)
{
      return ((digit == '0') || digit == '1');
}

bool Utility::IsDecimal(string sequence)
{
      for (int i = 0; i < sequence.length(); i++)
      {
            if (!isdigit(sequence[i]))
            return false;
      }
      return true;
}

bool Utility::IsHex(string sequence)
{
      for (int i = 0; i < sequence.length(); i++)
      {
            if (!isxdigit(sequence[i])) return false;
      }
      return true;
}

bool Utility::IsBinary(string sequence)
{
      for (int i = 0; i < sequence.length(); i++)
      {
            if (!(sequence[i] == '0' || sequence[i] == '1')) return
                  false;
      }
      return true;
}
```

A.1.8. utility.h

```
#ifndef _UTILITY_H_
#define _UTILITY_H_

#include <string>

using std::string;

class Utility
{
      private:

            /*
             * returns the hexadecimal symbol representing the integer if
             * the integer is less than 16, 0 if not
             */
             static char Symbol(int);

      public:

            /*
             * converts a 16-bit integer into its hexadecimal-string,
             * little endian representation that is made up of a
             * specified number of bits (8 or 16)
             */
             static string Int16ToHexString(int integer, int bits);

            /*
             * converts a 16-bit integer into its binary-string,
             * little endian representation that is made up of a
             * specified number of bits (8 or 16)
             */
             static string Int16ToBinaryString(int integer, int bits);

            /*
             * converts a hexadecimal-string, little endian
             * representation of a 16-bit integer into its numerical
             */equivalent
             static short int HexStringToInt16(string hexString);



            /*
             * converts a binary-string, little endian representation
             * of a 16-bit integer into its numerical equivalent
             */
             static short int BinaryStringToInt16(string binString);

            /*
             * checks if the character is a binary digit
             */
             static bool IsBinaryDigit(char digit);
```

```
            /*
             * checks if the string represents a decimal number
             */
             static bool IsDecimal(string decString);

            /*
             * checks if the string represents a hexadecimal number
             */
             static bool IsHex(string hexString);

            /*
             * checks if the string represents a binary number
             */
             static bool IsBinary(string binString);
};

#endif
```

A.1.9. fourier.h

```
# include <iostream>
# include <complex>

using namespace std;

class Fourier
{
            complex<double> data_array[1024];
            int block_length;

      public:
            void set(int*, int);
            void magnitude(double*);
            void FFT_function(void);

      private:
            void bit_reverse_function(void);

};


void Fourier::set(int data[],int length)
{
      delete[] data_array;
      block_length=length;

      for (int i=0; i<block_length; i++)
      data_array[i]=complex <double>(data[i], 0);
}
```

```
void Fourier::bit_reverse_function()
{
      int index = 0,  k = 0;
      complex<double> temp;
            for (int i = 0; i < block_length; i++)
      {
            if (index > i)
            {
            //Swapping of the data contents
            temp = data_array[index];
            data_array[index] = data_array[i];
            data_array[i] = temp;
            }
            k = block_length/2;
            while (index >= k)
            {
                  index = index - k;
                  k = k/2;
                  if (k == 0 || index == 0)
                        break;
            }
            index = index + k;
      }
}

void Fourier::FFT_function()
{
      // call bit reverse function
      bit_reverse_function();

      // indices for the DFT
      int k=0, index=0;
      int mmax=1, istep, sign=-1;

      // parameters for the computation of the twiddle factor
      double  theta=0.0, ur=0.0,ui=0.0, wcos=0.0,wsin=0.0;

      while (block_length > mmax)
      {
            istep = 2 * mmax;
            theta = 3.141592654/(float)(mmax);
            ur = 1.0; ui = 0.0;
            wcos = cos(theta);
            wsin = (float) sign * sin(theta);

            for (int j = 0; j < mmax; j ++)
            {
                  //declaration of the twiddle factor in complex form
                  complex<double> twiddle_factor(ur,ui);
                  for (k = j; k < block_length; k += istep)
                  {
                        index = k + mmax;
                        complex<double> twiddleTemp =
                              data_array[index]*twiddle_factor;
```

```
                    //calculation of the DFT's
                    data_array[index] = data_array[k] - twiddleTemp ;
                    data_array[k] = data_array[k] + twiddleTemp ;
                    }

                    //calculation for the twiddle factor
                    double wtemp;
                    wtemp = ur * wcos - ui * wsin;
                    ui = ui * wcos + ur * wsin;
                    ur = wtemp;
               }
          mmax = istep;
       }
}

void Fourier::magnitude(double mag[])
{
      // scaled computation of the magnitude of the complex fourier
            transform   .
      double scale = (1000/30000000);
      for (int i=0; i<block_length; i++)
      mag[i]=((abs(data_array[i]))*scale);
}
```

Bibliography

Baterina, Anna Veronica, Macalintal, Jeric. *xmcs_player_and_recorder.zip*. Internet. Available from http://groups.yahoo.com/group/admudsp/files/, accessed July 2010.

Decimation in Time FFT Algorithms. Internet. Available from http://www.ee.caltech.edu/EE/Courses/EE32b/handouts/FFT.pdf, accessed July 2010.

Fast Fourier Transform (FFT). Internet. Available from

http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html,accessed July 2010.

Fast Fourier Transform. Internet. Available from

http://www.mathcs.org/java/programs/FFT/FFTInfo/c12-2.pdf, accessed July 2010.

List. Internet. Available from

http://www.cplusplus.com/reference/stl/list/, accessed August 2010.

Koga, Dan. *Extended wxDev-C++ Tutorial*. 2009. Internet. Available from http://wxdsgn.sourceforge.net/?q=node/11, accessed 2 Aug 2010

Programming with wxDev-C++. 2006. Internet. Available from http://www.physics.ohio-state.edu/~ntg/780/downloads/programming_with_wxDec-C++.pdf, accessed 13 Aug 2010

Smart, Julian, Roebling, Robert, Zeitlin, Vadim. *Wx_wxdc*. Internet. Available from http://docs.wxwidgets.org/, accessed August 2010.

What are monophonic, polyphonic and true tone ringtones? 2009. Internet. Available from http://www.cellutips.com/what-are-monophonic-polyphonic-and-true-tone-ringtones/, accessed 10 October 2010.